

# 第1章 计算机与 C++ 编程简介

## 教学目标

- 了解计算机科学的基本概念
- 熟悉不同类型的编程语言
- 了解典型 C++ 程序的开发环境
- 用 C++ 编写简单的计算机程序
- 使用简单输入与输出语句
- 熟悉基本数据类型
- 使用算术运算符
- 了解算术运算符的优先级
- 编写简单的判断语句

## 1.1 简介

欢迎来到 C++ 的世界！我们将努力带给你一个信息丰富、充满趣味和富于挑战的学习经历。C++ 是一种较难的语言，通常只传授给有经验的程序员，因此本书在 C++ 教材中独具一格：

- 适合很少或没有编程经验的技术方面的人员
- 适合需要深入钻研 C++ 的熟练的程序员

一本书怎样适应两类读者呢？本书始终强调通过实践证明的结构化编程（structured programming）和面向对象编程（object-oriented programming）技术编写清晰的程序。非程序员从一开始就要养成良好的编程习惯。我们尽量以清晰而直接的方式编写程序。本书附有大量插图，更重要的是，本书提供大量实用的 C++ 程序，并显示这些程序在计算机上运行时产生的输出结果。所有 C++ 特性都是在完整、可工作的 C++ 程序环境中介绍的，我们称其为“有生命力的代码”。所有这些例子都可以从我们的 Web 站点 [www.deitel.com](http://www.deitel.com) 中下载，也可以通过本书所配的交互式光盘《C 与 C++ 多媒体教室（第二版）》中取得。多媒体教室的特性见本书最后的说明。多媒体教室中还包含本书一半练习的解答，包括简单解答、小型程序和许多完整项目。

本书的前五章介绍计算机基础、计算机编程和 C++ 计算机编程语言。参加我们课程的新学员告诉我，第 1 章到第 5 章的材料为今后学习 C++ 的高级知识打下了坚实基础。熟练的程序员可以快速浏览前五章，然后阅读本书其余部分对 C++ 的有深度、有挑战性的介绍。

许多熟练的程序员告诉我们，他们很欣赏我们对结构化编程的处理。他们通常用 C 或 Pascal 结构化编程语言进行编程，但由于没有系统地学习结构化编程，因此常常无法用这些语言编写出最佳代码。学习本书前几章介绍的结构化编程知识后，有助于改进使用 C 或 Pascal 语言编程的风格。因此，无论你是新手还是熟练的程序员，这里的信息都是充实、有趣并且具有一定深度的。

大多数人对计算机能做的工作略有所知,利用本书可以学会如何指示计算机做这些工作。软件 (software, 即编写的指令, 命令计算机完成操作并做出判断) 可以控制计算机 (通常称为硬件, hardware)。C++ 是当今最常用的软件开发语言之一。本书介绍的 C++ 版本已经过美国国家标准协会 (ANSI, American National Standards Institute) 和国际标准化组织 (ISO, International Standards Organization) 的标准化, 这个 ANSI/ISO C++ 草案标准已经批准为全球标准。

计算机的应用领域在不断增加。在这个成本稳步攀升的时代, 计算成本却在急速下降, 因为硬件和软件技术都在飞速发展。25年前装满整个房间、价值上百万美元的计算机如今已经缩小到比手指甲还小的芯片, 而且只要几美元。具有讽刺意味的是, 硅是地球上最丰富的资源之一, 是普通砂子的主要组成部分。硅芯片技术使计算技术如此经济, 如今全世界已经有大约2亿台通用计算机在使用中, 其应用涉及商业、工业、政府和个人生活。这个数字在近年内必将翻番。

本书将使读者面临几个挑战。几年前, 人们学习第一个编程语言时可能只要学习 C 或 Pascal, 但实际上还需要学习 C 和 C++, 为什么呢? 因为 C++ 包括 C 语言和其他更多的内容。

几年前人们只需学习结构化编程 (structured programming), 而今则既要学习结构化编程, 又要学习面向对象编程 (object-oriented programming), 因为面向对象是今后10年最关键的编程方法。本课程要建立和使用许多对象 (object), 但是这些对象的内部结构最好用结构化编程方法建立。另外, 操作对象的方法最好也用结构化编程方法来表达。

另一个介绍两种方法的原因是, 目前有大量 C++ 系统是从 C 语言系统移植过来的, 还有大量的所谓“C 语言遗留代码”。C 语言已经使用 20 余年, 近年来用得越来越多。人们学习 C++ 之后就会发现, C++ 比 C 语言强得多, 因此通常会转到使用 C++。他们会将遗留的系统移植到 C++, 这是个相对简单的过程。然后, 他们开始用各种 C++ 对 C 语言的增强特性改进其编写的类 C 语言程序的风格。最后, 他们开始利用 C++ 面向对象编程的功能, 真正了解这种语言的全部好处。

编程语言中的一个有趣现象是, 大多数厂家都推出 C/C++ 产品组合而不是提供分开的产品。这样, 用户可以继续使用 C 语言编程, 适当时候再逐渐过渡到 C++。

C++ 已经成为实现语言的首选, 但它能在第一门编程课程中介绍吗? 我们认为可以。五年前, 当人们用 Pascal 作为第一门编程课程时, 我们遇到过类似挑战。我们编写了《C How To Program》, 如今全世界几百所大学都在使用《C How To Program》第二版, 使用这本教材的课程和使用 Pascal 作为第一门编程课程一样有效。其间没有明显的差别, 只是学生学习的兴趣更高, 因为他们知道工作中要使用的是 C 语言而不是 Pascal 语言。学习 C 语言的学生还能更快地学习 C++ 和新的 Internet 语言——Java。

本书前五章介绍 C++ 中的结构化编程方法、C++ 的“C 语言部分”和“C++ 对 C 语言的改进”, 然后要介绍 C++ 面向对象编程, 但我们不想等到第6章再介绍面向对象编程, 因此前五章每一章都有一节“有关对象的思考”, 介绍面向对象编程的基本概念和术语。第6章“类与数据抽象”将开始用 C++ 生成对象并编写面向对象的程序。

第1章分为三部分, 第一部分介绍计算机基础和计算机编程。第二部分立即开始编写一些简单 C++ 程序, 第三部分介绍有关对象的思考。

下面要开始富有挑战和回报的旅程了。学习过程中, 如果想与我们联系, 可以给我们发电子邮件:

deitel@deitel.com

或浏览我们的 Web 站点:

<http://www.deitel.com>

我们将立即答复。希望大家喜欢学习《C++ 大学教程》，还可以使用本书的交互式光盘版本《C 与 C++ 多媒体教室（第二版）》，详见本书最后的说明。

## 1.2 什么是计算机

计算机 (computer) 是能以人的几百万甚至几十亿倍速度进行计算并作出逻辑判断的设备。例如,今天的许多个人计算机每秒钟可以进行几亿次加法运算。操作台式计算器的人要几十年才能算出的数值,强大的个人计算机只要一秒钟即可计算完毕(注意:你怎么知道这个人加对了没有?你怎么知道计算机做得是否正确?)。如今,最快的超级计算机 (supercomputer) 每秒钟可以进行几千亿次加法运算,是成百上千的人花一整年时间才能完成的计算工作。每秒钟万亿条指令的计算机已经能在研究实验室中工作。

计算机在一组指令控制下处理数据 (data), 这组指令称为计算机程序 (computer program)。这些计算机程序指导计算机按顺序进行计算机程序员 (computer programmer) 指定的一组操作。

构成计算机系统的各种设备 (如键盘、屏幕、鼠标、磁盘、内存、光盘和处理器) 称为硬件。计算机上运行的计算机程序称为软件。几年来,硬件成本已经大幅下降,使个人计算机更加平民化。但是,随着程序员开发了许多越来越强大、越来越复杂的应用程序,而软件开发技术却进步不大,因而使软件开发成本不断上升。本书介绍通过成熟的软件开发方法减少软件开发成本,即结构化编程、自上而下逐步完善、功能化以及面向对象编程。

## 1.3 计算机组成

不管外观如何不同,每个计算机都可以看成由六个逻辑单元 (logical unit) 或部分组成,即:

1. 输入单元 (input unit), 这是计算机的“接收”部分,从各种输入设备接收信息 (数据和计算机程序),并将这些信息放到其他单元中,使信息得以处理。如今大多数信息都是通过键盘和鼠标设备输入计算机。将来大多数信息也许可以通过语音输入或扫描图形而获得。
2. 输出单元 (output unit), 这是计算机的“发送”部分。将计算机处理过的信息送到不同输出设备中,向计算机外部提供所需的信息。如今计算机输出的大多数信息是通过屏幕显示、书面打印或用于控制其他设备。
3. 内存单元 (memory unit), 这是计算机中快速访问、低容量的“库存”部分。它保存通过输入单元输入的信息,以便在需要时立即提供这些信息进行处理。内存单元保存处理的信息,直到输出单元将信息放到输出设备中。内存单元也称为内存或主内存 (memory 或 primary memory)。
4. 算术/逻辑单元 (arithmetic and logic unit, ALU), 这是计算机中的“生产”部分,负责进行加、减、乘、除等运算,包含判断机制,例如可以让计算机比较内存单元中的两个项目,确定其是否相等。
5. 中央处理单元 (central processing unit, CPU), 这是计算机中的“管理”部分,是计算机的协调员,负责管理其他部分的操作。CPU 告诉输入单元何时将信息读取到内存单元中,告诉 ALU 何时利用内存单元中的信息进行计算,告诉输出单元何时将内存单元中的信息发送到指定的输出设备中。

6. 辅助存储单元 (secondary storage unit), 这是计算机长期的高容量“库存”部分, 其他单元不是经常使用的程序或数据通常放在辅助存储单元 (如磁盘) 中, 直到几小时、几天、数月甚至几年后才需要。访问辅助存储单元中的信息要比访问主内存中的信息慢得多。辅助存储单元的单位成本比主内存的单位成本低得多。

## 1.4 操作系统的变革

早期计算机一次只能完成一个任务或作业 (task 或 job), 这种计算机操作通常称为单用户批处理 (batch processing)。计算机一次运行一个程序, 成组或成批地处理数据。在这些早期系统中, 用户利用穿孔卡片将作业提交到计算机中心, 通常要等待几小时或几天之后才能得到打印输出。

称为操作系统 (operating system) 的软件系统可以帮助用户更方便地使用计算机。早期操作系统能管理作业之间的顺利过渡, 使得计算机操作员在作业之间切换的时间减到最少, 从而增加计算机处理的工作量或吞吐量 (throughput)。

随着计算机的功能越来越强大, 单用户批处理机制显然不能有效地利用计算机资源, 因此应该让许多任务或作业共享计算机资源, 以达到更好地利用资源, 这种方法称为多道程序设计 (multiprogramming)。多道程序系统涉及多个作业在计算机上“同时”操作, 计算机在竞争资源的作业之间共享资源。在早期多道程序操作系统中, 用户还是要通过穿孔卡片将作业提交到计算机中心, 几小时或几天之后才能得到打印输出。

20 世纪 60 年代, 计算机界和大学的几个研究小组提出了分时 (timesharing) 操作系统。分时是多道程序的特殊情况, 用户通过终端 (terminal) 访问计算机, 终端是带有键盘和屏幕的典型设备。在典型的分时操作系统中, 可能有几十甚至几百个用户同时共用计算机。计算机实际上并不是同时运行所有用户, 而是运行一个用户的一小段作业, 然后转入运行下一个用户的一小段作业。计算机的速度非常快, 每秒钟可以为每个用户服务多次, 使得用户的程序看上去是在同时运行。分时的好处之一是用户能立即收到响应, 而不必像原先的计算方式需要等待很长时间。

## 1.5 个人计算、分布式计算与客户/服务器计算

1977 年, Apple 计算机公司使个人计算 (personal computing) 得以普及。最初, 拥有一台计算机只是爱好者的梦想, 随着它的价格不断降低, 人们可以购买供个人或办公使用的计算机。1981 年, 世界上最大的计算机厂家 IBM 公司推出了 IBM 个人计算机 (IBM Personal Computer)。一夜之间, 个人计算机遍布公司、企业和政府机关。

然而这些计算机只是“独立”的个体, 各自做自己的工作, 要通过磁盘复制来共享信息 (通常称为暗联网)。尽管早期个人计算机不够强大, 不能同时服务于多个用户, 但这些机器可以链接在计算机网络中, 可以接入组织内的局域网 (Local Area Network, LAN), 还可以通过单位内部的电话线完成链接。这样就在组成化计算中出现了分布式计算 (distributed computing) 结构, 其处理不是在某个中央计算机上进行, 而是由分布于网络中的机器完成。个人计算机已经足够强大, 能够处理个人用户的计算要求并处理电子信息传递等基本通信任务。

如今, 最强大的个人计算机已经可以和十年前几百万美元的机器相媲美。最强大的台式计算机 (称为工作站, workstation) 对个人用户提供了大量的功能。在网络上, 有些计算机向遍布整个网络的客户提供数据存取服务, 这些计算机称为文件服务器 (file server)。通过这种方式, 在网络上共



享信息很容易,因而产生了客户-服务器结构。C和C++已经成为编写操作系统、计算机网络和分布式客户/服务器应用程序软件的首选编程语言。如今最常见的操作系统如UNIX、Microsoft 的基于Windows 系统和IBM 的OS/2 都提供了本节介绍的功能。

## 1.6 机器语言、汇编语言和高级语言

程序员用各种编程语言编写指令,有些是计算机直接理解的,有些则需要中间翻译(translation)的步骤。如今使用的计算机语言有几百种,可以分为三大类:

1. 机器语言
2. 汇编语言
3. 高级语言

任何计算机只能直接理解本身的机器语言(machine language)。机器语言是特定计算机的自然语言,由计算机的硬件设计定义。机器语言通常由一系列数字组成(最终简化为0和1),让计算机一次一个地执行最基本的操作。机器语言非常繁琐,下面的机器语言程序将工龄工资和基础工资相加,并把结果保存在工资总额中:

```
+1300042774
+1400593419
+1200274027
```

随着计算机越来越普及,机器语言编程对大多数程序员显然太慢、太繁琐。程序员不用计算机直接理解的一系列数字,而是用类似英文缩写的助记符来表示计算机的基本操作,这些助记符构成了汇编语言(assembly language)。称为汇编器(assembler)的翻译程序以计算机的速度将汇编语言程序转换为机器语言。下列汇编语言程序也是工龄工资和基础工资相加,并将结果保存在总工资中,但要比相应的机器语言清晰得多:

```
LOAD    BASEPAY
ADD     OVERPAY
STORE   GROSSPAY
```

尽管这种代码对于人们一目了然,但计算机却无法理解,必须先翻译为相应的机器语言。

随着汇编语言的出现,计算机的使用迅速增加,然而即使是最简单的任务,也需要许多条指令才能完成。为了加速编程过程,人们开发了高级语言(high-level language),用一条语句完成大量任务。称为编译器(compiler)的翻译程序将高级语言程序变为相应的机器语言。高级语言使程序员能够编写更像英语的指令,可以包含常用的数学符号。将工龄工资和基础工资相加,并把结果保存在总工资中,可以用下列高级语言程序:

```
grossPay = basePay + overTimePay
```

显然,从程序员角度看,高级语言比机器语言和汇编语言都要强得多。C和C++是最强大最广泛使用的高级语言。

将高级语言程序编译为相应的机器语言的过程可能需要大量时间。解释器(interpreter)程序可以直接执行高级语言程序,而不必先将这些程序编译成相应的机器语言。尽管编译程序的执行速

度比解释程序更快,但解释器在程序开发环境中更常用,因为增加新特性和纠正错误时经常需要重新编译程序。一旦程序开发完成,编译版本的运行最有效。

## 1.7 C语言与C++的历史

C++是从C语言演变而来的,而C语言又是从两个编程语言BCPL和B演变而来的,BCPL是Martin Richards于1967年开发的,用于编写操作系统软件和编译器。Ken Thompson在他的B语言中大量采用BCPL的特性,并用B语言在DEC PDP-7计算机上生成了UNIX操作系统的早期版本(1970年在贝尔实验室)。BCPL和B都是“无类型”语言,每个数据项在内存中占一个“字”(word)长,如果要数据项作为整数或实数处理,编程的工作量会很大。

C语言是从B语言演变而成的,由贝尔实验室的Dennis Ritchie开发,最初于1972年在DEC PDP-11计算机上实现。C语言使用了BCPL和B的许多重要概念,同时增加了数据类型和其他特性。C语言最初作为UNIX操作系统的开发语言而闻名于世。如今,大多数操作系统都是用C/C++写成的。二十多年来,C语言已经遍布在大多数计算机上。C语言是硬件无关的,只要仔细设计,就可以编写能移植到大多数计算机上的C语言程序。

到20世纪70年代末期,C语言演变成现在所谓的“传统C”、“经典C”或“Kernighan/Ritchie C”。1978年Prentice Hall公司出版了Kernighan和Ritchie合作的著作《The C Programming Language》,引起了人们对C语言的广泛关注(见参考文献Ke78)。

C语言在各种不同类型计算机(有时称为硬件平台)上的普及导致了許多变形。它们虽然相似,但通常互不兼容。对需要为不同平台编写可移植程序的开发人员,这是个严重问题,显然需要有个标准的C语言版本。1983年,美国国家计算机与信息处理标准委员会(X3)成立了X3J11技术分会,目的是提供无歧义性且与机器无关的语言定义。1989年推出了这种语言标准。ANSI与国际标准化组织(ISO)合作,在全球范围内将C语言标准化,1990年推出了联合标准文档,称为ANSI/ISO 9899:1990。这个文档可以从ANSI获得副本。1988年推出的Kernighan和Ritchie著作的第二版体现了该版本(称为ANSI C),这也是目前全世界使用的版本(见参考文献Ke88)。

### 可移植性提示 1.1

由于C语言是标准化、硬件无关、广为使用的语言,因此用C语言编写的应用程序通常只要稍作修改或不经修改即可在多种不同的计算机系统中运行。

C++是C语言的扩展,是20世纪80年代初由贝尔实验室的Bjarne Stroustrup开发的。C++的许多特性是从C语言中派生的,但更重要的是,它提供了面向对象编程(object-oriented programming)的功能。

软件业正在酝酿一场革命,最终目标是更快、更正确、更经济地建立软件,新的、更强大的软件需求迫在眉睫。对象(object)实际上是模拟实际项目的可复用软件组件(component)。软件开发人员发现,利用模块化、面向对象的设计和实现方法与过去结构化编程方法相比较,可以使软件开发小组的生产率更高。面向对象编程的优势在于更容易理解、纠正和修改。

许多面向对象的语言也纷纷涌现,包括最著名的由Xerox的Palo Alto研究中心(PARC)开发的Smalltalk。Smalltalk是纯粹的面向对象的语言,其所有的编程元素都是“对象”。C++则是一种“混合型语言”,可以用C语言方式、面向对象方式或兼用两种方式进行编程。1.9节将介绍基于C/C++的新语言——Java。

## 1.8 C++ 标准库

C++ 程序由类 (class) 和函数 (function) 组成。可以用多个小的软件模块构成 C++ 程序, 但大多数 C++ 程序员会利用 C++ 标准库中已有的类和函数来编程。这样, C++ “世界” 中实际要学习两方面的知识, 第一是学习 C++ 语言本身, 第二是学习如何利用 C++ 标准库中现有的类和函数 (本书将介绍许多类和函数)。Plauger (见参考文献 P192) 的著作是程序员必读的, 可以帮助程序员深入了解 C++ 中包括的 ANSI C 语言库函数, 了解如何实现这些库函数, 还可以了解如何用库函数编写可移植代码。标准库函数通常由编译器厂家提供。许多独立软件供应商 (independent software vendor, ISV) 也提供各种专用类库。

### 软件工程视点 1.1

使用构件块方法 (building block approach) 生成程序, 而不要事事从 0 开始。尽量利用现有程序块, 这称为软件复用 (software reuse), 是面向对象编程的核心。

### 软件工程视点 1.2

C++ 编程中通常使用下列构件块: C++ 标准库中的类和函数, 自己生成的类和函数和各种常见的非 C++ 标准库中的类和函数。

自己生成类和函数的优点在于知道其如何工作, 可以检查 C++ 代码; 缺点是要花大量时间及精力来设计、开发和维护这些类或函数, 使其正确、有效地运行。

### 性能提示 1.1

利用标准库函数和类而不用自己的对应版本可以提高软件性能, 因为这些软件经过认真编写, 能保证有效操作。

### 可移植性提示 1.2

利用标准库函数和类而不用自己的对应版本可以提高软件性能, 因为几乎所有 C++ 版本都包括这些软件。

## 1.9 Java、Internet 与万维网

这是 C++ 程序员应当高兴的时刻, 整个计算机行业都为新的 ANSI/ISO 草案标准最终得到批准而感到兴奋。

1995 年 5 月, Sun 公司宣布推出新的 Java 编程语言。Sun 公司是基于 UNIX 的高性能工作站的领导厂家, 一直强调计算机网络的重要性。Java 是基于 C/C++ 的语言, 加入了许多面向对象编程的特性。Sun 公司提供了基本 Java 软件、文档、教程和演示, 可以在 Web 站点 [www.javasoft.com](http://www.javasoft.com) 免费取得。Java 具有大量的类库, 包括支持多媒体、网络、图形、数据库访问、分布式计算等软件组件。

Java 最吸引人的属性之一是它的可移植性, 可以在一台计算机上编写 Java 程序, 然后在任何支持 Java 的计算机上运行 (目前大多数常见计算机系统都支持 Java)。这对软件开发人员特别具有吸引力, 他们可以不再为不同类型的计算机系统开发或维护不同的软件版本。为不同类型的计算机系统开发和维护不同的软件版本是非常费时、费力的, 使得独立软件供应商只能生产常见系统平台中使用的软件, 如 Microsoft Windows。如今, 基于 Java 的软件应用程序可以在所有常用的 Microsoft Windows 版本中和各种 UNIX、Macintosh 及 OS/2 等大多数流行的系统平台中运行。

我们体会到 Java 在学校和企业客户中的重要性, 因此编写了几本 Java 教材和基于交互式光盘的学习软件包。这个语言变化得很快, 因此我们在第一版《Java How to Program》推出后仅 11 个月又推出了其第二版。C++ 要成熟得多 (是 1980 年创建的), 因此《C++ 大学教程》在第一版推出三年半之后才推出第二版, 以配合 ANSI/ISO C++ 草案标准的推广。

## 1.10 其他高级语言

高级语言有数百种, 但被广泛采用的只有少数几种。FORTRAN (FoRmula TRANslator) 是 1954 到 1957 年之间由 IBM 公司开发的, 在需要复杂数学计算的科学和工程项目中应用较多。FORTRAN 仍然在工程领域广为使用。

COBOL (COmmon Business Oriented Language) 是 1959 年由计算机制造商、政府和工业企业计算机用户开发的。COBOL 擅长于需要精确和有效地操作大量数据的商业应用。因此, 大量的企业软件是用 COBOL 编写的。

Pascal (由 Niklaus Wirth 教授设计) 是与 C 语言同期出现的, 在学术界应用较多, 下一节将详细介绍 Pascal 语言。

## 1.11 结构化编程

20 世纪 60 年代, 许多大型软件的开发遇到了严重困难。常常推迟软件计划, 因而使成本大大超过预算, 而且最终产品也不可靠。人们开始认识到, 软件开发是项复杂的活动, 比原来所预想的要复杂得多。20 世纪 60 年代的研究结果是结构化编程 (structured programming) 的出现, 用规定的方法编写程序比非结构化编程能产生更清晰、更容易测试/调试以及更容易修改的程序。本书的第 2 章将介绍结构化编程原理。第 3 章到第 5 章则会开发多种结构化程序。

结构化编程研究的一个更具体结果是 1971 年 Niklaus Wirth 教授推出了 Pascal 语言。Pascal 语言是用 17 世纪著名数学家和哲学家巴雷斯·帕斯卡 (Blaise Pascal) 的名字命名的, 常用于教学中讲解结构化编程, 因而很快成为了大学中受欢迎的语言。但是这个语言缺乏在商业、工业和政府应用程序中所需要的许多特性, 因此没有被大学以外的环境所接受。

Ada 语言是在 20 世纪 70 年代和 80 年代初由美国国防部资助开发的。在此之前, 国防部的导弹命令与控制软件系统是由几百种不同语言生成的, 国防部要求用一种语言来完成大多数工作。Ada 以 Pascal 为基础, 但最终结构与 Pascal 大相径庭。这个语言是根据著名诗人 Lord Byron 的女儿 (Ada Lovelace) 的名字命名的。Ada Lovelace 在 19 世纪初编写了世界上第一个计算机程序, 用于 Charles Babbage 设计的分析机引擎的计算设备。Ada 的一个最重要功能是多任务 (multitasking), 程序员可以使多个活动任务并行发生。我们要介绍的其他常用高级语言 (包括 C/C++) 通常只让程序员编写一次只有一个活动任务的程序。

## 1.12 典型 C++ 环境基础

C++ 系统通常由几个部分组成: 程序开发环境、语言和 C++ 标准库。下面介绍图 1.1 所示的典型 C++ 环境。

C++ 程序通常要经过 6 个阶段 (如图 1.1), 即编辑 (edit)、预处理 (preprocess)、编译 (compile)、连接 (link)、装入 (load) 和执行 (execute)。这里主要介绍典型 UNIX C++ 系统 (注意, 本书的程

序不经修改或稍作修改即可在大多数当前的 C++ 系统中运行, 包括 Microsoft Windows 系统)。如果当前使用的不是 UNIX 系统, 可以参看系统手册或向老师请教如何在相应环境中完成这些工作。

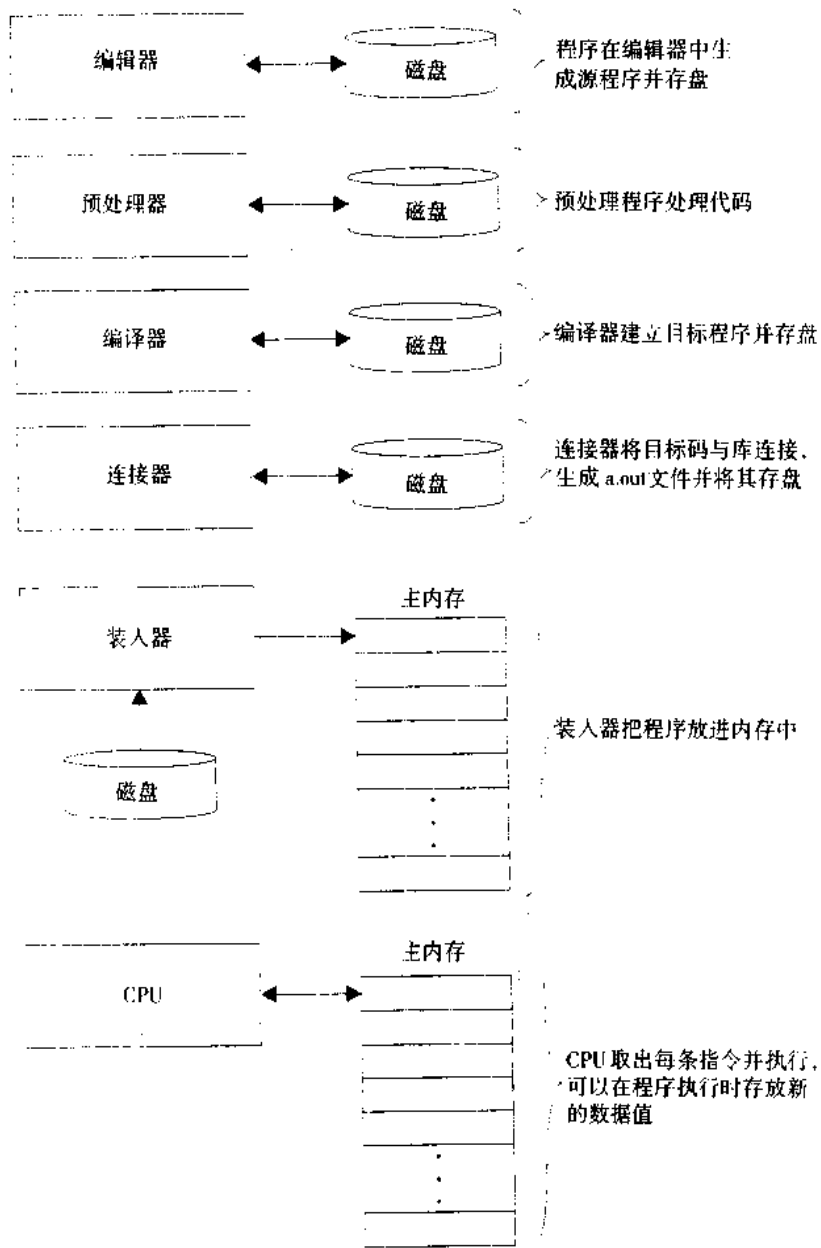


图 1.1 典型 C++ 环境

第一个阶段是编辑文件, 这是用编辑器程序 (editor program) 完成的。程序员用编辑器输入 C++ 程序, 并进行必要的修改, 然后将程序存放在磁盘之类的辅助存储设备中。C++ 程序文件名通常以 .cpp、.cxx 或 .C 的扩展名结尾 (注意 C 为大写), 详见 C++ 环境文档中对文件名扩展的说明。UNIX 系统中两个广泛使用的编辑器是 vi 和 emacs。个人计算机上的 Borland C++ 和 Microsoft Visual C++ 等 C++ 软件包都有自己的编辑器, 它们与编程环境紧密集成。这里, 我们假设读者已经知道如何编辑程序。

随后的阶段是程序员发出编译 (compile) 程序的命令。编译器将 C++ 程序翻译为机器语言代码 (也称为目标码)。在 C++ 系统中, 预处理程序在编辑器翻译阶段开始之前自动执行。C++ 预处理器采用预处理指令 (preprocessor directive) 表示程序编译之前要进行的某些操作。这些操作通常包括在要编译的文件中包括其他文本文件和进行各种文本替换。前面几章将介绍最常见的预处理指令, 所有预处理指令的详细介绍见第 17 章“预处理器”。编辑器在将程序翻译为机器语言代码之前调用预处理器。

下一个阶段是连接。C++ 程序常常引用其他地方定义的函数, 如标准库中或特定项目的程序员使用的专用库。C++ 编译器产生的目标码通常包含由于缺少一些内容而造成的“空穴”, 连接器 (linker) 将目标码与这些默认功能的代码连接起来, 建立执行程序映像 (不再缺少任何代码)。在典型的 UNIX 系统中, 编译和连接 C++ 程序的命令是 CC。要编译和连接程序 welcome.C, 在 UNIX 提示符下键入:

```
CC welcome.C
```

并按 Enter 键 (或 Return 键)。如果程序编译和连接正确, 则产生文件 a.out。这就是 welcome.C 程序的执行程序映像。

再下一个阶段是装入。执行之前, 要先把程序放进内存中, 这是由装入器 (loader) 完成的, 装入器读取磁盘中执行程序的映像文件, 并将其放进内存中。

最后, 计算机在 CPU 控制下逐条指令地执行程序。要在 UNIX 系统中装入并执行程序, 可在 UNIX 提示符下键入 a.out 并按 Return 键。

程序不是一次就能够运行。上述每个阶段都可能因为各种错误而失败。例如, 可能除数为 0 (计算机上的非法操作, 和算术运算中一样), 这会使计算机打印出错信息。然后, 程序员需返回编辑阶段, 进行必要的修改并继续其余阶段, 确定修改之后能否顺利工作。

#### 常见编程错误 1.1

如果除数为 0 之类的错误在程序运行时发生, 则称这类错误为运行时错误 (run-time error) 或执行时错误 (execution-time error)。除数为 0 通常是个致命错误, 会使程序立即终止, 无法完成工作。非致命错误能让程序运行完毕, 但会产生错误结果 (注意, 在有些系统中, 除数为 0 并不是致命错误, 详见系统的说明文档)。

C++ 中的大多数程序都要输入/输出数据。有些 C++ 函数的输入来自 cin (标准输入流), 通常是键盘, 但也可以连接其他输入设备。数据通常输出到 cout (标准输出流), 一般是计算机屏幕, 也可以是其他设备。程序打印结果, 通常是指在屏幕上显示结果。数据也可以输出到其他设备, 如磁盘和硬拷贝打印机。还有称为 cerr 的标准错误流 (standard error stream), 通常连接屏幕, 用于显示错误消息。用户经常把普通输出数据 (即 cout) 路由到非屏幕设备, 而让 cerr 输出到屏幕, 可以立即通知用户发生错误。

### 1.13 C++ 与本书的一般说明

C++ 是个复杂的语言。熟练的 C++ 程序员有时以能够编写一些稀奇古怪的小程序为荣, 这可不是好的编程习惯, 因为这样会使程序更难阅读、更难测试和调试, 也很难根据需求改变而改变。本书面向初学的程序员, 因此我们强调清晰性。下面是第一个编程技巧。

**编程技巧 1.1**

C++ 程序应以简单和直接的方式编写，称为 KIS (“keep it simple”，保持简单)，不要使用不常用的方法任意地扩大程序。

本书提供了许多编程技巧，帮助读者养成良好习惯，编写更清晰、更易懂、更易维护、更易测试和调试的代码。这些技巧只是个指导原则，读者完全可以选择自己喜欢的编程风格。我们还介绍常见编程错误（在程序中要注意各种问题以避免这些错误）、性能提示（使程序的运行速度更快和使用更少内存的方法）、可移植性提示（帮助编写不经修改或稍作修改即可在其他计算机上运行的程序）、软件工程视点（影响和提高软件系统总体结构的概念和思想，特别是大型软件系统）和测试与调试提示（测试程序以找出和消除缺陷的提示）。

前面曾介绍过，C 和 C++ 是可移植语言，C 和 C++ 编写的程序可以在许多不同计算机上运行。可移植性是个重要目标。ANSI C 语言标准文档（见参考文献 An90）中列出了大量移植性问题，还有介绍移植性问题的专著（见参考文献 Ja89 和 Ra90）。

**可移植性提示 1.3**

尽管可以编写可移植程序，但不同的 C 和 C++ 编译器和不同计算机的许多问题使移植性难以实现。C 或 C++ 写成的程序并不一定是可移植程序。程序员通常需要直接涉及不同的编译器和不同的计算机。

前面介绍了 ANSI/ISO C++ 草案标准文档的演变过程和检查了其完整性与准确性。但 C++ 是个丰富的语言，其中有些细节和高级课题是我们所没有介绍的。如果需要 C++ 的详细技术信息，可以阅读这个文档的最新草案。从下列 Web 站点可以取得草案：

<http://www.cygnum.com/misc/wp/>

我们列出了关于 C++ 和面向对象编程的大量文献与图书目录，还列出了 C++ 资源附录，包含许多与 C++ 和面向对象编程有关的 Web 站点。

当前 C++ 版本的特性与旧版 C++ 不兼容，因此本文中的有些程序也许无法在旧版的 C++ 编译器中工作。

**编程技巧 1.2**

阅读所用的 C++ 版本的手册。经常翻阅这些手册，能够知道 C++ 的丰富特性并正确使用这些特性。

**编程技巧 1.3**

使用计算机和编译器有助于学习 C++。如果阅读所用 C++ 版本的手册之后还不知道 C++ 工作的特性，可以试用一个小的测试程序（test program），看看其如何工作。设置编译器选项为最大警告（maximum warning）。注意编译程序时出现的每个消息，并纠正问题，消除这些消息。

## 1.14 C++ 编程简介

C++ 语言提供了计算机程序设计的结构化和规则化方法。我们现在要介绍 C++ 编程，并用几个例子演示 C++ 的许多重要特性，每个例子一次分析一条语句。第 2 章介绍 C++ 中结构化编程的详细处理，然后到第 5 章一直使用结构化编程方法。第 6 章开始介绍 C++ 面向对象编程，由于面向对象编程在本书的核心重要性，因此前五章各有一节“有关对象的思考”。这些小节介绍面向对象编程的概念和实例，让读者设计和实现面向对象的 C++ 程序。

## 1.15 简单程序：打印一行文本

C++使用非程序员可能感到奇怪的符号。我们首先介绍一个简单程序：打印一行文本。程序及其屏幕输出如图 1.2。

这段程序演示了 C++ 语言的几个重要特性。我们详细介绍程序的每一行。

```
// Fig. 1.2: fig1_02.cpp
// A first program in C++
```

以//开头，表示该行其余部分是注释语句（comment）。程序员插入注释语句用来说明程序和提高程序的可读性。注释语句还可以帮助其他人阅读和理解程序。在运行程序时注释语句并不使计算机采取任何操作。C++ 编译器忽略注释语句，不产生任何机器语言目标码。注释语句“first program in C++”只是描述该程序的用途。以//开头的说明语句称为单行注释语句（single-line comment），因为该注释语句在行尾结束（注意：C++ 程序员也可以用 C 语言注释语句样式，即注释语句以/\* 开头，以\*/结束）。

```
1 // Fig. 1.2: fig01_02.cpp
2 // A first program in C++
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome to C++!\n";
8
9     return 0;    // indicate that program ended successfully
10 }
```

**输出结果：**

Welcome to C++!

图 1.2 文本打印程序

### 编程技巧 1.4

每段程序应以注释语句开头，描述该程序的用途。

下列语句：

```
#include <iostream.h>
```

是条预处理指令（preprocessor directive），是发给 C++ 预处理器的消息。预处理器先处理以#开头的一行语句之后再编译程序。这一行预处理指令告诉预处理器要在程序中包括输入/输出流头文件 `iostream.h` 的内容，应该在任何使用 C++ 式输入/输出流从键盘输入数据或向屏幕输出数据的程序中包括这个文件。图 1.2 中的程序向屏幕输出数据，`iostream.h` 文件的内容将在稍后详细介绍。注意，最新 ANSI/ISO C++ 草案标准实际上指定 `iostream.h` 和其他标准头文件不需要后缀 `.h`，如 `iostream`。我们在本书余下部分继续使用旧式头文件，因为许多编译器还不支持最新的 ANSI/ISO C++ 草案标准。1.20 节将再次介绍这个例子，演示如何使用新式头文件。

### 常见编程错误 1.2

如果从键盘输入数据或向屏幕输出数据的程序中没有包括 `iostream.h` 文件，则编译器会发出一个错误消息。



下列语句:

```
int main()
```

是每个C++程序都包含的语句。main后面的括号表示main是个程序基本组件,称为函数(function)。C++程序包含一个或几个函数,其中有且只有一个main函数。即使main不是程序中的第一个函数,C++程序通常都从函数main开始执行。main左边的关键字int表示main返回一个整数值。我们将在第3章深入介绍函数时介绍函数返回值的含义。目前只要在每个程序的main函数的左边包括关键字int即可。

左花括号({)应放在每个函数体(body)开头,对应右花括号(})应放在每个函数体结尾。下列语句:

```
cout << "Welcome to C++!\n";
```

让计算机在屏幕上打印引号之间的字符串(string)。整个行称为语句(statement),包括cout、<<运算符、字符串"Welcome to C++!\n"和分号(;)。每条语句应以分号(又称为语句终止符)结束。C++中的输出和输入是用字符流(stream)完成的,这样,执行上述语句时,将字符流"Welcome to C++!"发送到标准输出流对象(standard output stream object)cout,通常cout将其输出到屏幕。第11章"C++输入/输出流"中将详细介绍cout。

运算符<<称为流插入运算符(stream insertion operator)。执行这个程序时,运算符右边的值(右操作数)插入输出流中。右操作数数字通常按引号中的原样直接打印。但注意字符\n不在屏幕中打印。反斜杠(\)称为转义符(escape character),表示要输出特殊字符。字符串中遇到反斜杠时,下一个字符与反斜杠组合,形成转义序列(escape sequence)。转义序列\n表示换行符(newline),使光标(即当前屏幕位置的指示符)移到屏幕中下一行开头。表1.3列出了常用转义序列。

#### 常见编程错误1.3

省略语句末尾的分号是个语法错误,语法错误使编译器无法识别一个语句。编译器通常会发出错误消息,帮助程序员找到和纠正错误。语法错误即违反了语言规则,语法错误也称为编译错误,因为它们是在编译阶段出现。

下列语句:

```
return 0; // indicate that program ended successfully
```

放在每个main函数末尾。C++的关键字return是退出函数的几种方式之一。main函数末尾使用return语句时,数值0表示程序顺利结束。第3章将详细介绍函数和解释包括这个语句的原因。目前只要记住在每个程序中都要包括这个语句,否则在某些程序中编译器会产生警告消息。

右花括号(})表示main函数结束。

转义序列	说明
\n	换行符,使屏幕光标移到屏幕中下一行开头
\t	水平制表符,使屏幕光标移到下一制表位
\r	回车符,使屏幕光标移到当前行开头,不移到下一行
\a	警告,发出系统警告声音
\\	反斜杠,打印反斜杠符
\"	双引号,打印双引号

图1.3 常用转义序列

**编程技巧 1.5**

许多程序员让函数打印的最后一个字符为换行符(\n)。这样可以保证函数使屏幕光标移到屏幕中下一行开头。这种习惯能促进软件复用,这是软件开发环境中的关键目标。

**编程技巧 1.6**

将每个函数的整个函数体在定义函数体的花括号中缩排一级,可使程序的函数结构更明显,使程序更易读。

**编程技巧 1.7**

确定一个喜欢的缩排长度,然后一直坚持这个缩排长度。可以用制表符生成缩排,但制表位可能改变。建议用 1/4 英寸制表位或三个空格的缩排长度。

“Welcome to C++!”可用多种方法打印。例如,图 1.4 的程序用多条流插入语句,产生的程序输出与图 1.2 相同,因为每条流插入语句在上一条语句停止的位置开始打印。第一个流插入语句打印“Welcome”和空格,第二条流插入语句打印同一行空格后面的内容。C++ 允许以多种方式表达语句。

一条语句也可以用换行符打印多行,如图 1.5。每次在输出流中遇到\n转义序列时,屏幕光标移到下一行开头。要在输出中得到空行,只要将两个\n放在一起即可,如图 1.5。

```
1 // Fig. 1.4: fig01_04.cpp
2 // Printing a line with multiple statements
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome ";
8     cout << "to C++!\n";
9
10    return 0;    // indicate that program ended successfully
11 }
```

**输出结果:**

Welcome to C++!

图 1.4 用多条流插入语句打印一行

```
1 // Fig. 1.5: fig01_05.cpp
2 // Printing multiple lines with a single statement
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome\nto\n\nC++!\n";
8
9     return 0;    // indicate that program ended successfully
10 }
```

**输出结果:**

Welcome  
to  
  
C++!

图 1.5 用一条流插入语句打印多行

## 1.16 简单程序：两个整数相加

下一个程序用输入流对象cin和流读取运算符>>取得用户从键盘中输入的两个整数,计算这两个值的和,并将结果用cout输出。程序及其输出如图1.6。

```
1 // Fig. 1.6: fig01_06.cpp
2 // Addition program
3 #include <iostream.h>
4
5 int main()
6 {
7     int integer1, integer2, sum;           // declaration
8
9     cout << "Enter first integer\n";       // prompt
10    cin >> integer1;                       // read an integer
11    cout << "Enter second integer\n";      // prompt
12    cin >> integer2;                       // read an integer
13    sum = integer1 + integer2;             // assignment of sum
14    cout << "Sum is " << sum << endl;     // print sum
15
16    return 0;    // indicate that program ended successfully
17 }
```

**输出结果：**

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

图 1.6 两个整数相加

注释语句：

```
// Fig. 1.6: fig01_06.cpp
// Addition program
```

指定文件名和程序用途。C++ 预处理指令：

```
#include <iostream.h>
```

将 iostream.h 头文件的内容放进程序中。

前面曾介绍过,每个程序从main函数开始执行。左花括号表示main函数体开头,相应右花括号表示main函数体结束。下列语句：

```
int integer1, integer2, sum;    // declaration
```

是个声明 (declaration)。integer1、integer2 和 sum 是变量 (variable) 名。变量是计算机内存中的地址,存放程序使用的值。这个声明指定变量integer1、integer2 和 sum 的数据类型为 int,表示这些变量保存整数值,如 7、-11、0、31914。所有变量都应先声明名称和数据类型之后才能在程序中使用。几个相同类型的变量可以在同一声明或多个声明中声明。我们可以一次只声明一个变量,但一次声明多个同类型变量更加简练。

**编程技巧 1.8**

有些程序员喜欢一行只声明一个变量，这样可以在每个声明后面插入注释语句。

稍后要介绍数据类型 `float`（定义实数，即带小数点的数，如 3.4、0.0、-11.19）和 `char`（定义字符型数据，变量 `char` 只能保存一个小写字母、一个大写字母、一个数字或一个特殊字符，如 `x`、`$`、`7`、`*` 等等）。

**编程技巧 1.9**

在每个逗号（`,`）后面加上空格，使程序更易读。

变量名是任何有效标识符（`identifier`）。标识符是一系列由字母、数字和下划线（`_`）组成的字符串，不能以数字开头。C++ 是区分大小写的，因此 `al` 和 `Al` 是不同的标识符。

**可移植性提示 1.4**

C++ 允许任意长度的标识符，但系统和 C++ 版本可能限制标识符的长度不超过 31 个字符，以保证移植性。

**编程技巧 1.10**

选择有意义的变量名能使程序更清楚，只要阅读程序就可以比较容易理解程序，而不必阅读手册或使用其他注释语句。

**编程技巧 1.11**

避免用下划线和双下划线开头的标识符，因为 C++ 编译器内部使用这类名称。这样可以防止与编译器选择的名称冲突。

变量声明可以放在函数的任何位置，但变量声明必须放在程序中使用变量之前。例如在图 1.6 所示的程序中，如果不用一条语句声明三个变量，也可以分别声明。下列声明：

```
int integer1;
```

可以放在下列语句之前：

```
cin >> integer1;
```

下列声明：

```
int integer2;
```

可以放在下列语句之前：

```
cin >> integer2;
```

下列声明：

```
int sum;;
```

可以放在下列语句之前：

```
sum = integer1 + integer2;
```

**编程技巧 1.12**

可执行语句之间的声明之前最好留一行空格，这样能使声明更明显，程序更清晰。

**编程技巧 1.13**

如果喜欢把声明放在函数开头，应在这些声明的结尾与该函数中的执行语句开始之间留一行空格，将这些声明与该函数中的执行语句分开。

下列语句:

```
cout << "Enter first integer\n"; //prompt
```

在屏幕上打印字符串 Enter first integer (也称为字符串直接量 (string literal) 或直接量 (literal)), 将光标移到下一行开头。这个消息称为提示 (prompt), 提醒用户进行特定操作。上述语句表示 cout 得到字符串 "Enter first integer\n"。

下列语句:

```
cin >> integer1; //read an integer
```

用输入流对象 cin 和流读取运算符 >> 取得键盘中的值。利用流读取运算符和 cin 从标准输入流读取输入 (通常是键盘输入)。上述语句表示 cin 提供 integer1 的值。

计算机执行上述语句时, 等待用户输入变量 integer1 的值。用户输入整数值并按 Enter 键 (或 Return 键), 将数值发送给计算机。然后计算机将这个数 (值) 赋给变量 integer1。程序中后面引用 integer1 时都使用这个值。

cout 和 cin 流对象实现用户与计算机之间的交互。由于这个交互像对话一样, 因此通常称为对话式计算 (conversational computing) 或交互式计算 (interactive computing)。

下列语句:

```
cout << "Enter second integer\n"; //prompt
```

在屏幕上打印 "Enter second integer" 字样, 然后移到下一行开头。这个语句提示用户进行操作。下列语句:

```
cin >> integer2; //read an integer
```

从用户取得变量 integer2 的值。

赋值语句:

```
sum = integer1 + integer2; // assignment of sum
```

计算变量 integer1 和 integer2 的和, 然后用赋值运算符 (assignment operator) "=" 将结果赋给变量 sum。这个语句表示 sum 取得 integer1 加 integer2 的值。大多数计算都是在赋值语句中进行的。"=" 运算符和前面的 "+" 运算符称为二元运算符 (binary operator), 因为它们有两个操作数。对于 "+" 运算符, 两个操作数是 integer1 和 integer2。而对于 "=" 运算符, 两个操作数是 sum 和表达式 integer1 + integer2 的值。

#### 编程技巧 1.14

二元运算符两边要放上空格, 这样能使运算符更明显, 程序更易读。

下列语句:

```
cout << "Sum is" << sum << endl; // print sum
```

打印字符串 "Sum is" 和变量 sum 的数值, 加上称为流操纵算子的 endl (end line 的缩写)。endl 输出一个换行符, 然后刷新输出缓冲区, 即在一些系统中, 输出暂时在机器中缓存, 等缓冲区满时再打印到屏幕上, endl 强制立即输出到屏幕上。

注意, 上述语句输出多种不同类型的值, 流插入运算符知道如何输出每个数据。在一个语句中使用多个流插入运算符称为连接 (concatenating)、链接 (chaining) 或连续使用流插入操作。这样,

就不必用多条输出语句输出多个数据。

计算可以在输出语句中进行。可以将上述语句合二为一：

```
cout << "Sum is" << integer1 + integer2 << endl;
```

从而不需要变量 `sum`。

右花括号告诉计算机到达了函数 `main` 的结尾。

C++ 的一个强大特性就是用户可以生成自己的数据类型（详见第 6 章），然后可以告诉 C++ 如何用 `>>` 和 `<<` 运算符输入或输出这种类型的值（称为运算符重载，见第 8 章）。

## 1.17 内存的概念

`integer1`、`integer2` 和 `sum` 等变量名实际上对应于计算机内存中的地址（location）。每个变量都有名称（name）、类型（type）、长度（size）和值（value）。

在图 1.6 所示的加法程序中，执行下列语句时：

```
cin >> integer1;
```

用户输入的值放在 C++ 编译器为 `integer1` 指定的内存地址中。假设用户输入 `integer1` 的值 45，则计算机将 45 放在地址 `integer1` 中，如图 1.7。

无论何时将新数值放入内存地址，这个值将取代该地址中原有的值，并删除前一个值。

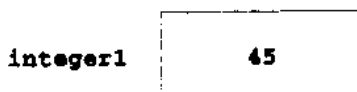


图 1.7 内存中的值

运行前面提到的加法程序，当执行下列语句时：

```
cin >> integer2;
```

假设用户输入值 72，则计算机将 72 放在地址 `integer2` 中，如图 1.8。注意编译器不一定将这两个地址指定为内存中的相邻地址。

程序取得 `integer1` 和 `integer2` 值后，它将这两个值相加，并将和放在变量 `sum` 中。下列语句：

```
sum = integer1 + integer2;
```

进行加法运算，同时也删除一个值，即把 `integer1` 和 `integer2` 的和放进地址 `sum` 中时，`sum` 原有的值丢失。计算 `sum` 之后，内存如图 1.9。注意 `integer1` 和 `integer2` 的值和计算之前一样，虽然这些值在计算机进行计算时使用，但并不删除。因此，从一个内存地址读取数值时，这个过程是非破坏性的。

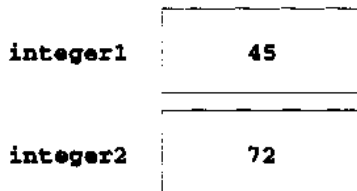


图 1.8 输入两个变量值之后的内存值

<b>integer1</b>	<b>45</b>
<b>integer2</b>	<b>72</b>
<b>sum</b>	<b>117</b>

图 1.9 计算之后的内存值

## 1.18 算术运算

大多数程序都要进行算术运算。算术运算符见图 1.10, 注意这里使用了许多代数中没有使用的符号。星号 (\*) 表示乘法、百分号 (%) 表示求模 (modulus), 将在稍后介绍。图 1.10 所示的算术运算符都是二元运算符, 即这些运算符取两个操作数。例如, 表达式 “integer1 + integer2” 包含二元运算符 “+” 和两个操作数 integer1 和 integer2。

C++ 操作	算术运算符	代数表达式	C++ 表达式
加	+	$f+7$	<code>f + 7</code>
减	-	$p-c$	<code>p - c</code>
乘	*	$bm$	<code>b * m</code>
除	/	$x/y$ 或 $x/y$ 或 $x \div y$	<code>x / y</code>
求模	%	$r \bmod s$	<code>r % s</code>

图 1.10 算术运算符

整除 (即除数和被除数均为整数) 取得整数结果。例如, 表达式  $7/4$  得 1, 表达式  $17/5$  得 3。注意, 整除结果忽略分数部分, 不用取整。

C++ 提供求模 (modulus) 运算符 “%”, 即求得整除的余数。求模运算符是个整数运算符, 只能使用整数操作数。表达式  $x\%y$  取得  $x$  除以  $y$  的余数, 这样,  $7\%4$  得到 3,  $17\%5$  得 2。后面几章将介绍求模运算符许多有趣的应用, 如确定一个数是否为另一个数的倍数 (确定一个数为奇数或偶数是这个问题的一个特例)。

### 常见编程错误 1.4

对非整型操作数使用求模运算符是个语法错误。

C++ 中的算术表达式应以直线形式在计算机中输入。这样,  $a$  除以  $b$  应输入为 “ $a/b$ ”, 使所有常量、变量和运算符放在一行中。编译器通常不接受下列代数符号:

$$\frac{a}{b}$$

但有些特殊专用软件包支持复杂数学表达式更自然的表示方法。

C++ 表达式中括号的使用和代数表达式中相同。例如, 要将  $a$  乘以  $b+c$  的和, 可以用:

$$a * (b + c)$$

C++ 中算术运算符的运算顺序是由运算符优先级规则确定的, 与代数中相同:

1. 括号中的表达式先求值，程序员可以用括号指定运算顺序。括号具有最高优先级，对于嵌套括号，由内层向外层求值。
2. 乘法、除法、求模运算优先。如果表达式中有多个乘法、除法、求模运算，则从左向右求值。乘法、除法、求模的优先级相同。
3. 然后再进行加法和减法。如果表达式中有多个加法和减法，则从左向右求值。加法和减法的优先级相同。

运算符优先级保证C++按正确顺序采用运算符。从左向右求值指的是运算符的结合律(associativity)，也有一些运算符的结合律是从右向左。图 1.11 总结了运算符优先级规则，引入其他 C++ 运算符时，这个表可以扩充。详细的运算符优先级请参见附录。

运算符	运算	求值顺序
()	括号	最先求值，如果嵌套括号，则先求最内层表达式的值。如果同一层有几对括号，则从左向右求值
*, /, 或 %	乘除求模	其次求值。如果有多个，则从左向右求值
+ 或 -	加减	最后求值。如果有多个，则从左向右求值

图 1.11 算术运算符优先级

下面用几个表达式说明运算符优先级规则。每个例子都列出代数表达式和对应的C++表达式。

下例求五个值的算术平均值：

$$\text{代数: } m = \frac{a+b+c+d+e}{5}$$

C++: `m = (a+b+c+d+e)/5;`

括号是必需的，因为除法的优先级比加法高，要把整个和 (a+b+c+d+e) 除以 5，如果不加括号，则 `a+b+c+d+e/5` 取值为：

$$a+b+c+d+\frac{e}{5}$$

下例是直线的方程：

$$\text{代数: } y = mx + b$$

C++: `y = m * x + b;`

不需要括号，乘法优先于加法，因此先乘后加。

下例包含模 (%)、乘、除、加、减运算：

$$\text{代数: } z = pr\%q + w/x - y$$

C++: `z = p * r % q + w / x - y;`

⑥ ① ② ④ ③ ⑤

语句下面的圆圈数字表示C++采用运算符的顺序。乘法、求模和除法首先从左向右求值(结合律为从左向右)，因为它们的优先级高于加法和减法。然后进行加法和减法运算，也是从左向右求值。

并不是有多对括号的表达式都包含嵌套括号。例如下列表达式不包含嵌套括号：

$$a * (b + c) + c * (d + e)$$



这些括号在同一层。

要更好地了解运算符优先级规则，考虑二次多项式的求值：

$y = a * x * x + b * x + c;$   
 ⑥    ①    ②    ④    ③    ⑤

语句下面的圆圈数字表示 C++ 采用运算符的顺序。C++ 中没有指数算术运算符，因此我们把  $x^2$  表示为  $x * x$ ，稍后会介绍标准库函数 `pow`（指数）。由于 `pow` 所要的数据类型有一些特殊情况，因此放到第 3 章再介绍。

假设变量  $a$ 、 $b$ 、 $c$ 、 $x$  初始化如下： $a=2$ 、 $b=3$ 、 $c=7$  和  $x=5$ 。图 1.12 演示了上述二次多项式的运算符优先级。

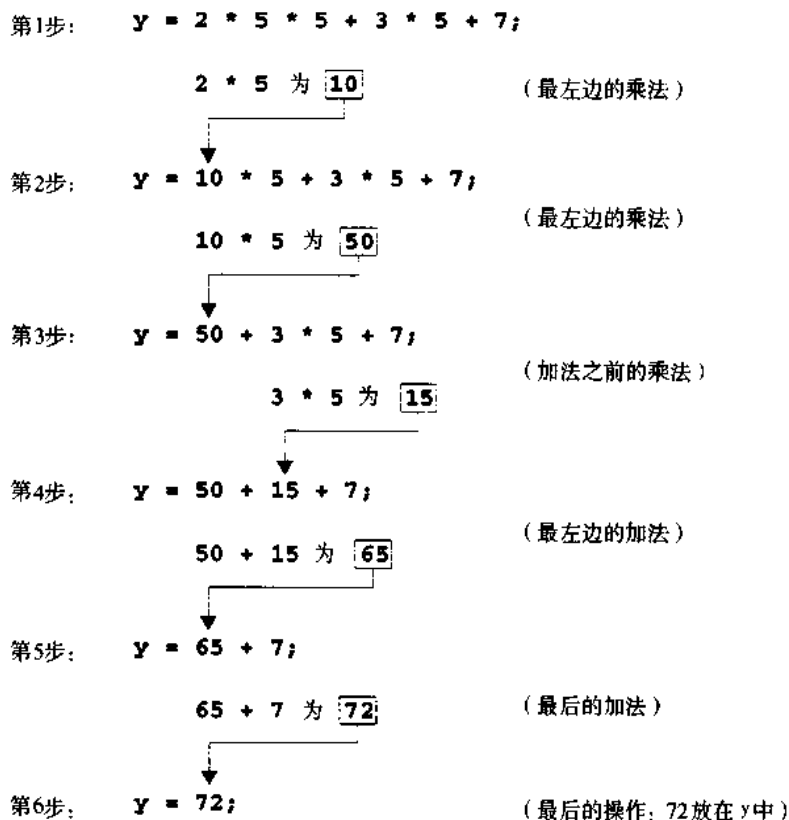


图 1.12 二次多项式的求值顺序

上述赋值语句可以加上多余的括号，使代码更清晰：

$y = (a * x * x) + (b * x) + c;$

#### 编程技巧 1.15

和代数中一样，可以在表达式中加上多余的括号，使代码更清晰，这些括号称为冗余括号。冗余括号常用于组合大表达式中的子表达式，使表达式更加清晰。

## 1.19 判断：相等与关系运算符

本节介绍简单的 C++ 的 `if` 结构，使程序根据某些条件的真假作出判断。如果条件符合，即为真 (`true`)，则执行 `if` 结构体的语句；如果不符合，即条件为假 (`false`)，则不执行语句，稍后将举例说明。

if结构中的条件可以用相等运算符 (equality operator) 和关系运算符 (relational operator) 表示, 如图 1.13 关系运算符具有相同的优先级, 结合律为从左向右。相等运算符的优先级也都相同, 但低于关系运算符的优先级, 结合律也为从左向右。

标准代数相等与关系运算符	C++ 相等与关系运算符	C++ 条件举例	C++ 条件含义
相等运算符			
=	==	x == y	x 等于 y
≠	!=	x != y	x 不等于 y
关系运算符			
>	>	x > y	x 大于 y
<	<	x < y	x 小于 y
≥	>=	x >= y	x 大于或等于 y
≤	<=	x <= y	x 小于或等于 y

图 1.13 相等与关系运算符

#### 常见编程错误 1.5

如果 ==、!=、>= 和 <= 运算符的符号对之间出现空格, 则会出现语法错误。

#### 常见编程错误 1.6

逆转 !=、>= 和 <= 运算符的符号顺序 (变为 =!、=> 和 =<) 通常会出现语法错误。将 != 写成 =! 有时不会出现语法错误, 但却会出现逻辑错误。

#### 常见编程错误 1.7

不要把相等运算符 == 与赋值运算符 = 混淆起来。相等运算符表示等于, 而赋值运算符表示取、取值或赋值。有人把相等运算符读作双等于。稍后会介绍, 把相等运算符 == 与赋值运算符的 = 混淆起来可能不会造成明显的语法错误, 但可能造成相当特殊的逻辑错误。

下例用六个 if 语句比较用户输入的两个数。如果其中任何一个 if 语句的条件成立, 则执行与该 if 相关联的输出语句。图 1.14 显示了这个程序和三个示例输出。

注意图 1.14 的程序连续使用流读取操作输入两个整数。首先将一个值读取到 num1 中, 然后将一个值读取到 num2 中。if 语句的缩排是为了提高程序的可读性。另外, 注意图 1.14 中每个 if 语句体中有一条语句。第 2 章将会介绍结构体中有多条语句的 if 语句 (将语句体放在花括号 “{}” 中)。

```

1 // Fig. 1.14: fig01_14.cpp
2 // Using if statements, relational
3 // operators, and equality operators
4 #include <iostream.h>
5
6 int main()
7 {
8     int num1, num2;
9
10    cout << "Enter two integers, and I will tell you\n"
11    << "the relationships they satisfy: ";
12    cin >> num1 >> num2;    // read two integers
13
14    if ( num1 == num2 )
15        cout << num1 << " is equal to " << num2 << endl;
16
17    if ( num1 != num2 )

```

```
18     cout << num1 << " is not equal to " << num2 << endl;
19
20     if ( num1 < num2 )
21         cout << num1 << " is less than " << num2 << endl;
22
23     if ( num1 > num2 )
24         cout << num1 << " is greater than " << num2 << endl;
25
26     if ( num1 <= num2 )
27         cout << num1 << " is less than or equal to "
28             << num2 << endl;
29
30     if ( num1 >= num2 )
31         cout << num1 << " is greater than or equal to "
32             << num2 << endl;
33
34     return 0;    // indicate that program ended successfully
35 }
```

**输出结果:**

```
Enter two integers, and I will tell you
The relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

图 1.14 使用相等和关系运算符

**编程技巧 1.16**

if语句的缩排可以提高程序的可读性，突出结构体。

**编程技巧 1.17**

程序中一行只放一条语句。

**常见编程错误 1.8**

if语句的条件后面的右括号之后紧接着分号通常是个逻辑错误（但不是语法错误）。分号使if结构体变成空的，不管条件是否为真，这个if结构都不进行任何操作。更糟糕的是，原先if结构体变成if结构后面的语句，不管条件是否为真，总是执行，通常会使程序产生错误结果。

注意图 1.14 中空格的用法。在 C++ 语言中，空白字符（如制表符、换行符和空格）通常被编译器忽略。因此，语句中可以根据程序员的爱好加上换行符和空格，但不能用换行符和空格分隔标识符。

**常见编程错误 1.9**

把 main 写成 ma in 是语法错误。

**编程技巧 1.18**

长语句可以分成多行。如果一条语句要分成多行，可以在分隔列表的逗号或长表达式的运算符后面断行。如果语句要分成两行或多行，后续行可以缩排。

图 1.15 显示了本章介绍的运算符优先级，运算符优先顺序从上向下递减。注意除赋值运算符之后的所有运算符结合律均为从左向右。加法是左结合的，因此表达式  $x+y+z$  求值为  $(x+y)+z$ 。赋值运算符是从右向左结合的，因此表达式  $x=y=0$  求值为  $x=(y=0)$ ，首先将 0 赋给  $y$ ，然后将赋值的结果 0 赋给  $x$ 。

运算符	结合律	类型
()	从左向右	括号
*     /     %	从左向右	乘
+     -	从左向右	加
<<     >>	从左向右	流插入/读取
<     <=     >     >=	从左向右	关系
==     !=	从左向右	等于
=	从右向左	赋值

图 1.15 运算符优先级和结合律

**编程技巧 1.19**

编写包含多个运算符的表达式时要参考运算符优先级，确保表达式中的运算符按所要求的运算。如果无法确定复杂表达式中的求值顺序，可以用括号强制顺序，就像代数表达式中一样。注意，有些运算符（如赋值运算符）是从右向左结合的，而不是从左向右。

前面介绍了 C++ 的许多特性，包括在屏幕上打印数据、从键盘输入数据、进行计算和作出判断。第 2 章利用这些知识介绍结构化编程，将会学到更多的缩排技巧，介绍如何指定和改变语句执行顺序（称为控制流）。

## 1.20 新型头文件与名字空间

本节是为使用支持 ANSI/ISO 草案标准的编译器用户提供的。草案标准指定了许多旧式 C++ 头文件的新名，包括 `iostream.h`，大多数新式头文件不再用扩展名 `.h`。图 1.16 改写图 1.2，演示新型头文件和两种使用标准库头文件特性的方法。

第 3 行：

```
#include <iostream>
```

演示新型头文件名语法。

第 5 行：

```
using namespace std;
```

指定用 `std` 名字空间（`namespace`），这是 C++ 中的新特性。名字空间可以帮助程序员开发新的软件组件而不会与现有软件组件产生命名冲突。开发类库的一个问题是类和函数名可能已经使用。名字空间能为每个新软件组件保持唯一的名称。

```
1 // Fig. 1.16: fig01_16.cpp
2 // Using new-style header files
3 #include <iostream>
4
5 using namespace std;
6
7 int main()
8 {
9     cout << "Welcome to C++!\n";
10    std::cout << "Welcome to C++!\n";
11
12    return 0;    // indicate that program ended successfully
13 }
```

**输出结果:**

```
Welcome to C++!
Welcome to C++!
```

图 1.16 使用新型头文件

C++ 草案标准中的每个头文件用名字空间 `std` 保证今后 C++ 标准库操作的每个特性是惟一的,不会与其他程序员开发的软件组件混起来,程序员不能用名字空间 `std` 定义新的类库。上述语句只是表示我们使用 C++ 标准库中的软件组件,如果要定义自己的类库,则可以将我们的所有类和函数放在名字空间 `deitel` 中,使我们的类库与所有其他公司的类库和 C++ 标准类库区别开来。

程序中出现“`using namespace std`”语句之后,就可以像第 9 行那样用 `cout` 对象将数值输出到标准输出流中。如果使用两个或几个类库,其中有的特性名称相同,则可能引起名称冲突。这时就要用名字空间来完全限定所用的名称,例如第 10 行的 `std::cout`:

```
std::cout << "Welcome to C++!\n";
```

`cout` 的完全限定名为 `std::cout`,如果全部用这种形式,虽然比较繁琐,但程序中第 5 行的“`using namespace std`”语句就没有必要了。`using` 语句可以在 C++ 标准库中使用每个名称的简写版本(或其他指定名字空间的名称),我们将在本书稍后详细介绍名字空间。目前并不是所有 C++ 环境都已经支持新的头文件命名格式。为此,我们在本书大部分地方使用旧式头文件,只在介绍 C++ 草案标准新特性时才使用新的头文件命名格式,使用新格式时将特别注明。

## 1.21 有关对象的思考

下面要开始介绍面向对象。面向对象是观察世界和编写计算机程序的自然方式。

那么,为什么不从一开始就介绍面向对象呢?我们为什么要把面向对象编程放到第 6 章再介绍呢?原因是我们要建立的对象是由各个结构化程序段组成的,因此先要介绍结构化编程基础。

前面五章先介绍结构化编程的传统方法。然后在每一章的最后介绍面向对象。第一章要介绍基本概念(如“有关对象的思考”)和术语(如“对象表达”)。第 2 章到第 5 章考虑更实质的问题,然后用面向对象设计(object-oriented design, OOD)技术攻克难题。我们将分析建立系统时的典型问题语句,确定实现系统需要哪些对象、对象需要哪些属性、表示哪些行为以及指定对象之间如何交互,以满足系统的需要。我们用实际问题而不是课堂例子来介绍这些内容,而且是在编写面向对象的 C++ 程序之前介绍。第 6 章就可以开始在 C++ 中编写面向对象的系统。

我们首先介绍一些面向对象的关键术语。看看实际生活中，对象无处不在，人、动物、植物、汽车、飞机、大楼、计算机都是对象，人们通过对象来思考问题。我们有较高的抽象（abstraction）能力，可以将屏幕图形作为人、飞机、树、山等对象，而不是各个颜色点（称为像素）。我们能看到沙滩而不是一粒粒沙子，森林而不是一棵棵树，房子而不是一块块砖。

我们经常把对象分为两类：活动对象与非活动对象。活动对象是运动的，可以活动和工作。而非活动对象（如毛巾）则不能做什么事，只能静止在某处。但所有这些对象却有一些共同之处，即具有尺寸、形状、颜色、重量等属性（attribute），并具有一些行为（behavior），如球可以滚动、跳动、收缩和膨胀；婴儿会哭、会睡、会爬、会跑、会眨眼；汽车能加速、刹车、转向；毛巾能吸水等等。

人们通过观察对象属性和行为而了解对象。不同对象可能有相似的属性和行为。例如，可以比较婴儿与成人，比较人与猩猩。汽车、卡车、货车和轻便小客车也有一些相似之处。

面向对象编程（object-oriented programming, OOP）用软件对象模拟实际对象。它利用类（class）关系，其中某一类的对象（如汽车类）有一些共同特性。它利用继承（inheritance）关系，还可以用多重继承（multiple inheritance）从现有类派生新类，并在新类中增加独特的特征。敞篷车类对象有一些汽车类的属性，但敞篷车的车顶可以打开、关闭。

面向对象编程可以更自然、更直观地浏览编程过程，即构造现实对象、其属性和行为。OOP还构造对象之间的通信，就像人与人之间可以互相发送消息一样（如军官命令士兵立正），对象也可以通过消息相互通信。

OOP将数据（属性）和函数（行为）封装（encapsulate）成对象（object），对象的数据和函数是密切联系的。对象具有信息隐藏（information hiding）属性，即对象虽然通过定义接口（interface）能够相互通信，但该对象通常不知道其他对象的实现方法，因为实现细节隐藏在对象内部。显然，我们可以很好地驾驶汽车而不需要知道发动机、传送系统和燃料系统内部如何工作。稍后将会介绍信息隐藏对于出色的软件工程是何等重要。

在C语言和其他过程式编程语言（procedural programming language）中，编程是面向操作的（action-oriented）；而在C++中，编程是面向对象的（object-oriented）。在C语言中，编程单位是函数（function）；而在C++中，编程单位是类（class），通过类可以最终实例化（即生成）对象。C++的类包含函数。

C语言程序员将精力更多地放在编写函数上。完成相同任务的一组操作组成了函数，而一些函数又组成了程序。C语言中的数据虽然很重要，但数据只是用于支持函数所要完成的操作。系统中的动词（verb）帮助C语言程序员确定一组函数，这组函数一起实现系统。

C++程序员则要集中考虑生成自己的用户自定义类型（称为类）。每个类包含数据和一组操作数据的函数。类的数据组件称为数据成员（data member），而类的函数组件称为成员函数（在其他面向对象编程语言中通常称为方法）。内部类型（如int）的实例称为变量，而用户自定义类型（即类）的实例称为对象。程序员用内部类型作为构造用户自定义类型的基本组件。C++中关注的重点是类（构成对象的类）而不是函数。系统中的名词（noun）帮助C++语言程序员确定一组类，由这些类生成系统的对象。对象的类就像房子的蓝图，我们可以用一张蓝图建造多个房子，同样也可以用一类生成多个对象。

软件打包成类时，这些类变成组件，可以在今后的软件系统中复用。就像房地产代理商告诉客户影响房地产价位的三个最主要因素是“地段、地段和地段”，同样，影响今后软件开发的三个最主要因素是“复用、复用和复用”。

事实上利用对象技术,今后大多数软件都是组合“标准化、可互换组件”的类而生成,本书介绍如何建立可复用类。每个新类都是宝贵的软件资源,可以用于加速和提高今后的软件开发速度。

本章介绍了一个C++控制结构——if结构。第2章将介绍六个其他控制结构,介绍如何组合控制结构与各种操作语句,形成结构化程序。我们将会发现只需要三种不同的控制结构,这些结构只要用两种不同组合方式就可以形成任何C++程序。第2章末尾我们将继续介绍面向对象,确定实际系统中需要实现的对象。

## 小结

- 计算机是一种设备,能够进行计算和逻辑判断,运算速度是人类速度的几百万甚至几十亿倍。
- 计算机在计算机程序的控制下处理数据。
- 构成计算机系统的各种设备(如键盘、屏幕、鼠标、磁盘、内存、光盘和处理单元)称为硬件。
- 计算机上运行的计算机程序称为软件。
- 输入单元是计算机的“接收”部分,如今大多数信息都是通过键盘和鼠标设备输入计算机。
- 输出单元,这是计算机的“发送”部分。计算机输出的大多数信息是通过屏幕显示或书面打印。
- 内存单元,这是计算机中的“库存”部分,也称为内存或主内存。
- 算术逻辑单元(ALU)进行计算和判断。
- 没有经常使用的程序或数据通常放在辅助存储单元(如磁盘)中,直到需要时再使用。
- 在单用户批处理中,计算机一次运行一个程序,成组或成批处理数据。
- 利用称为操作系统的软件系统可以更方便地使用计算机和取得更好的性能。
- 多道程序操作系统使多个作业在计算机上“同时”操作,这些作业共享计算机资源。
- 分时是多道程序的一种特殊情况,用户通过终端访问计算机,使用户的程序好像是同时运行。
- 在分布式计算中,组成化计算分布在网络上实际工作的站点。
- 服务器提供公用存储程序和数据,供网络中分布的客户计算机使用,从而出现了客户/服务器计算。
- 任何计算机只能直接了解其本身的机器语言。
- 机器语言通常包括数字字符串(最终简化为0和1),让计算机一次一个地进行最基本的操作。机器语言是机器相关的。
- 用类似英文缩写的助记符构成了汇编语言,人们用汇编器按计算机的速度将汇编语言程序转换为机器语言。
- 编译程序将高级语言程序变为相应的机器语言。高级语言使程序员能够编写更像英语的指令,可以包含常用的数学符号。
- 解释器程序可以直接执行高级语言程序而不必先将这些程序编译成相应的机器语言。
- 尽管编译程序的执行速度比解释程序更快,但解释器在程序开发环境中更常用,因为增加新特性和纠正错误时经常需要更新编译程序。程序开发完成后,编译版本的运行最有效。
- 可以用C/C++编写能移植到大多数计算机的程序。
- FORTRAN用于数学应用。
- COBOL主要用于需要精确和有效地操作大量数据的商业应用程序。

- 结构化编程用规定的方法编写比非结构化编程更清晰、更容易测试和调试、更容易修改的程序。
- Pascal 用于大学环境中讲解结构化编程。
- Ada 编程语言是由美国国防部资助开发的，以 Pascal 为基础。
- 多任务使程序员可以指定多个活动并行发生。
- C++ 系统通常由几个部分组成：程序开发环境、语言和 C++ 标准库。标准库函数本身不属于 C++ 语言，而是用于进行数学运算等常用操作。
- C++ 程序通常要经过六个阶段执行，即编辑、预处理、编译、连接、装入和执行。
- 程序员用编辑器输入 C++ 程序，并进行必要的修改。C++ 程序文件名通常以 .cpp、.cxx 或 .C 扩展名结尾。
- 编译器将 C++ 程序翻译为机器语言代码（也称为目标码）。
- C++ 预处理器采用预处理指令，通常包括在要编译的文件中包括其他文本文件和进行各种文本替换。
- 连接器将目标码与这些默认功能的代码连接起来，建立执行程序映像（不缺少内容）。在典型的 UNIX 系统中，编译和连接 C++ 程序的命令是 CC。如果程序编译和连接正确，则产生文件 a.out，这是该程序的执行程序映像。
- 装入器取出磁盘中的执行程序，并将其放进内存中。
- 计算机在 CPU 控制下一次一条指令地执行程序。
- 除数为 0 之类的错误在程序运行时发生，因此这类错误称为运行时错误。
- 除数为 0 通常是个致命错误，会使程序立即终止，无法完成工作。非致命错误能让程序运行完毕，但通常会产生错误结果。
- 有些 C++ 函数的输入来自 cin（标准输入流），通常输入来自键盘，但 cin 也可以连接其他设备。数据输出到 cout（标准输出流），通常输出到计算机屏幕，但 cout 也可以连接另一设备。
- 标准错误流 cerr 通常连接屏幕，用于显示错误消息。
- 不同 C++ 编译器和不同计算机的许多问题使移植性难以实现。
- C++ 提供面向对象编程功能。
- 对象实际上是可复用软件组件，构造实际中的项目。对象是由其“蓝图”（类）生成的。
- 单行注释语句以“//”开头。程序员插入注释语句用来说明程序和提高程序可读性。注释语句在运行程序时并不使计算机采取任何操作。
- “#include <iostream.h>”语句是个预处理指令。告诉预处理器要在程序中包括输入/输出流头文件。这个文件在编译使用 cin、cout 以及 >> 与 << 运算符的程序时需要。
- C++ 程序通常从 main 开始执行。
- 输出流对象 cout 通常连接屏幕，用于输出数据。多个数据项可以通过连接流插入运算符 (<<) 输出。输入流对象 cin 通常连接键盘，用于输入数据。多个数据项可以通过连接流读取运算符 (>>) 输入。
- 所有变量都应先声明名称和数据类型之后才能在程序中使用。
- C++ 变量名可以是任何有效标识符。标识符是一系列由字母、数字和下划线（\_）组成的字符串，不能以数字开头。C++ 允许任意长度的标识符，但系统和 C++ 版本可能限制标识符长度。
- C++ 是区别大小写的。
- 大多数计算都是在赋值语句中进行。



- 计算机内存中的每个变量都有名称、类型、长度和值。
- 数值放在内存地址时, 这个值取代该地址中原有的值, 前一个值被删除。
- 从一个内存地址读取数值时, 这个过程是非破坏性的, 只是读取一个副本, 原件保留。
- C++ 中算术运算符的运算顺序是由运算符优先级和结合律确定的。
- If 结构根据某些条件的真假值作出判断, if 结构的格式如下:

```
if (condition)
    statement;
```

- 如果条件符合, 即为 true (真), 则执行 if 结构体的语句; 如果不符合, 即条件为 false (假), 则不执行该语句。
- if 结构中的条件可以用相等运算符和关系运算符表示, 这些运算符的结果总是 “true” 或 “false”。
- 面向对象是观察世界和编写计算机程序的天然方式。
- 对象具有尺寸、形状、颜色、重量等属性, 并具有一些行为。
- 人们通过观察对象属性和行为而了解对象。
- 不同对象可能有相似的属性和行为。
- 面向对象编程 (OOP) 用软件对象模拟实际对象。它利用类关系, 其中某一类的对象有一些共同特性。它利用继承关系, 还可以用多重继承从现有类派生新类, 并在新类中增加独特的特征。
- 面向对象编程可以更自然、更直观地浏览编程过程, 即构造现实对象、属性和行为。
- OOP 可以通过消息在对象之间相互通信。
- OOP 将数据 (属性) 和函数 (行为) 封装成对象。
- 对象具有信息隐藏的属性, 即对象虽然通过定义接口能够相互通信, 但对象通常不知道其他对象的实现方法, 因为实现细节隐藏在对象内部。
- 信息隐藏对良好的软件工程非常重要。
- 在 C 语言和其他过程式编程语言中, 编程是面向操作。C 语言中的数据虽然很重要, 但数据只是用于支持函数要完成的操作。
- C++ 程序员则要集中考虑生成自己的用户自定义类型 (称为类), 每个类包含数据和一组操作数据的函数。类的数据组件称为数据成员, 而类的函数组件称为成员函数或方法。

## 术语

abstraction 抽象

action 操作

ANSI/ISO C

ANSI/ISO C++ draft standard

arithmetic and logic unit (ALU) 算术逻辑单元

arithmetic operators 算术运算符

assembly language 汇编语言

assignment operator (=) 赋值运算符

associativity of operators 运算符结合律

attributes of an object 对象属性

behaviors of an object 对象行为

binary operator 二元运算符

body of a function 函数体

C

C++

C++ standard library C++ 标准库

case sensitive 大小写相关

central processing unit (CPU) 中央处理单元

- cin object cin 对象  
clarity 清晰性  
class 类  
client/server computing 客户/服务器计算  
comment(//) 注释语句  
compile error 编译错误  
compile-time error 编译时错误  
compiler 编译器  
computer 计算机  
computer program 计算机语言  
condition 条件  
cout object cout 对象  
CPU 中央处理器  
crafting valuable classes 构造重要类  
data 数据  
data member 数据成员  
decision 判断  
declaration 声明  
distributed computing 分布式计算  
editor 编辑器  
encapsulation 封装  
equality operators 相等运算符  
    == "is equal to" 等于  
    != "is not equal to" 不等于  
escape character(\) 转义符  
escape sequence 转义序列  
execution-time error 运行时错误  
fatal error 致命错误  
file server 文件服务器  
flow of control 控制流  
fully qualified name (with namespaces) 完全限定名  
function 函数  
hardware 硬件  
high-level language 高级语言  
identifier 标识符  
if structure if 结构  
information hiding 信息隐藏  
inheritance 继承  
input device 输入设备  
input/output (I/O) 输入/输出  
integer(int) 整数  
integer division 整除  
interface 接口  
interpreter 解释器  
iostream  
iostream.h  
left-to-right associativity 从左向右结合律  
linking 连接  
loading 装入  
logic error 逻辑错误  
machine dependent 机器相关  
machine independent 机器无关  
machine language 机器语言  
main  
member function 成员函数  
memory 内存  
memory location 内存地址  
message 消息  
method 方法  
modeling 构造  
modulus operator(%) 求模运算符  
multiple inheritance 多重继承  
multiplication operator(\*) 乘法运算符  
multiprocessor 多处理器  
multiprogramming 多道程序处理  
multitasking 多任务  
namespace  
newline character(\n) 换行符  
nested parentheses 嵌套的括号  
non-fatal error 非致命错误  
object 对象  
object-oriented design(OOD) 面向对象设计  
object-oriented programming(OOP) 面向对象编程  
operand 操作数  
operator 运算符  
operator associativity 运算符结合律  
output device 输出设备  
parentheses 括号  
precedence 优先级  
preprocessor 预处理器  
primary memory 主内存  
procedural programming 过程式编程  
programming language 编程语言  
prompt 提示

relational operators 关系运算符	standard input object (cin) 标准输入对象
> "is greater than" 大于	standard output object (cout) 标准输出对象
< "is less than" 小于	statement 语句
>= "is greater than or equal to" 大于或等于	statement terminator (;) 语句终止符
<= "is less than or equal to" 小于或等于	std::cout
reserved words 保留字	string 字符串
"reuse, reuse, reuse" 复用、复用、复用	structured programming 结构化编程
right-to-left associativity 从右向左结合律	syntax error 语法错误
rules of operator precedence 运算符优先级规则	translator program 翻译器程序
run-time error 运行时错误	user-defined type 用户自定义类型
semicolon (;) 分号, 语句终止符	using namespace std;
software 软件	variable 变量
software asset 软件资源	variable name 变量名
software reusability 软件可复用性	variable value 变量值
standard error object (cerr) 标准错误对象	white-space characters 空白字符

## 自测练习

### 1.1 填空:

- 使个人计算普及的公司是\_\_\_\_\_。
- 使个人计算适用于企业和公司的计算机是\_\_\_\_\_。
- 计算机在称为计算机\_\_\_\_\_的指令集控制下处理数据。
- 计算机的六个关键逻辑单元是\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
- 本章介绍的三类语言是\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
- 将高级语言程序翻译为机器语言的程序称为\_\_\_\_\_。
- C语言以作为开发\_\_\_\_\_操作系统的语言著称。
- Wirth开发了\_\_\_\_\_语言, 用于大学中讲解结构化编程。
- 国防部开发的Ada语言具有\_\_\_\_\_功能, 可以指定并行处理多个活动。

### 1.2 关于C++程序环境的填空:

- C++程序通常用\_\_\_\_\_将程序输入计算机中。
- 在C++系统中, \_\_\_\_\_程序在编译器翻译阶段开始之前执行。
- \_\_\_\_\_程序组合编译器输出与各种库函数, 产生执行程序映像。
- \_\_\_\_\_程序将C++程序执行程序映像从磁盘传输到内存中。

### 1.3 填空:

- 每个C++程序从函数\_\_\_\_\_开始执行。
- 每个函数体以\_\_\_\_\_开始, 以\_\_\_\_\_结束。
- 每条语句以\_\_\_\_\_结束。
- 转义序列\n表示\_\_\_\_\_符, 使光标移到屏幕上一行开头。
- \_\_\_\_\_语句用于进行判断。

### 1.4 判断下列各题是否正确。如果不正确, 请说明原因。

- 程序执行时, 注释语句使计算机在屏幕上打印//之后的文本。

- b) `cout` 输出时转义序列 `\n` 使光标移到屏幕上下一行开头。
  - c) 所有变量都应先声明后使用。
  - d) 所有变量声明时要指定类型。
  - e) C++ 中变量 `number` 和 `NuMbEr` 是相同的。
  - f) 声明可以放在 C++ 函数体的任何地方。
  - g) 求模运算符 (%) 只能用于整数操作数。
  - h) 算术运算符 \*、/、%、+、- 的优先级相同。
  - i) C++ 程序要打印三行文本输出，就要有三个使用 `cout` 的输出语句。
- 1.5 编写下列 C++ 语句：
- a) 将变量 `c`、`thisIsAVariable`、`q76354` 和 `number` 声明为 `int` 类型。
  - b) 提示用户输入一个整数，提示消息后面是冒号 (:) 加空格，并将光标放在空格后面。
  - c) 读取用户键盘输入并将输入值存放在整型变量 `age` 中。
  - d) 如果变量 `number` 不等于 7，则打印 “The variable number is not equal to 7”。
  - e) 将消息 “This is a C++ program” 在一行中打印。
  - f) 将消息 “This is a C++ program” 在两行中打印，第一行在 C++ 处断行。
  - g) 将消息 “This is a C++ program” 在多行中打印，每个单词一行。
  - h) 打印消息 “This is a C++ program”，单词之间用制表符分开。
- 1.6 编写下列 C++ 语句 (或注释语句)：
- a) 表示程序计算三个整数的积。
  - b) 将变量 `x`、`y`、`z` 和 `result` 声明为 `int` 类型。
  - c) 提示用户输入三个整数。
  - d) 读取用户键盘输入的三个整数并将其存放在变量 `x`、`y`、`z` 中。
  - e) 计算三个整数的积并将结果赋给变量 `result`。
  - f) 打印 “The product is ” 加上变量 `result` 的值。
  - g) 从 `main` 返回，表示程序顺利结束。
- 1.7 用练习 1.6 编写的语句编写一个完整程序，计算三个整数的积。
- 1.8 判断并纠正下列语句中的错误：
- a) `if (c < 7 ) ;`  
`cout << "c is less than 7\n";`
  - b) `if (c == 7 )`  
`cout << "c is equal to or greater than 7\n";`
- 1.9 有关对象问题的填空：
- a) 人可以看电视屏幕和看到屏幕上的颜色点，也许可以看到三个人坐在会议室里，这种功能称为 \_\_\_\_\_。
  - b) 如果把汽车看成对象，则把车看成交通工具是车的属性还是行为 (选一个) \_\_\_\_\_。
  - c) 汽车可以加速、减速、左转、右转、向前、向后，这是汽车对象的 \_\_\_\_\_。
  - d) 新的类继承几个不同类的特征时，称为 \_\_\_\_\_ 继承。
  - e) 对象通过发送 \_\_\_\_\_ 相互通信。
  - f) 对象通过定义良好的 \_\_\_\_\_ 而相互通信。
  - g) 每个对象通常不能了解其他对象的实现细节，这个属性称为 \_\_\_\_\_。
  - h) 系统指定中的 \_\_\_\_\_ 帮助 C++ 程序员确定系统中要实现的类。

- i) 类的数据组件称为\_\_\_\_, 类的函数组件称为\_\_\_\_。  
j) 用户自定义类型的实例称为\_\_\_\_。

### 自测练习答案

- 1.1 a) Apple。b) IBM Personal Computer。c) 程序。d) 输入单元、输出单元、内存单元、算术与逻辑单元、中央处理单元、辅助存储单元。e) 机器语言、汇编语言、高级语言。f) 编译器。g) UNIX。h) Pascal。i) 多任务。
- 1.2 a) 编辑器。b) 预处理器。c) 连接器。d) 装入器。
- 1.3 a) main。b) 左花括号 ( { )、右花括号 ( } )。c) 分号。d) 换行。e) if。
- 1.4 a) 不正确。程序执行时, 注释语句不产生任何操作, 它们只是表示说明程序和提高程序可读性。  
b) 正确。  
c) 正确。  
d) 正确。  
e) 不正确。C++ 是区别大小写的, 两个变量是不同变量。  
f) 正确。  
g) 不正确。  
h) 不正确。运算符 \*、/、% 的优先级相同, 而运算符 +、- 的优先级较低。  
i) 不正确。一个输出语句可以用 cout 和多个 \n 转义序列打印多行文本。
- 1.5 a) `int c, thisIsAVariable, q76354, number;`  
b) `cout << "Enter an integer: ";`  
c) `cin >> age;`  
d) `if (number != 7 )`  
    `cout << "The variable number is not equal to 7\n";`  
e) `cout << "This is a C++ program\n";`  
f) `cout << "This is a C++\nprogram\n";`  
g) `cout << "This\nis\na\nC++\nprogram\n";`  
h) `cout << "This\tis\ta\tC++\tprogram\n";`
- 1.6 a) `// Calculate the product of three integers`  
b) `int x, y, z, result;`  
c) `cout << "Enter three integers: ";`  
d) `cin >> x >> y >> z;`  
e) `result = x * y * z;`  
f) `cout << "The product is " << result << endl;`  
g) `return 0;`
- 1.7 `// Calculate the product of three integers`  
`#include <iostream.h>`  
  
`int main()`  
`{`  
    `int x, y, z, result;`  
    `cout << "Enter three integers:";`  
    `cin >> x >> y >> z;`  
    `result = x * y * z;`  
    `cout << "The product is " << result << endl;`  
`}`

```
    return 0;  
}
```

- 1.8 a) 不正确: if 语句条件的右括号后面有分号。

纠正: 删除右括号后面的分号。注意: 这个错误的结果是不管 if 语句是否为真, 都执行输出语句。右括号后面的分号导致空语句。下一章将介绍空语句。

- b) 不正确: 没有关系运算符  $=>$ 。

纠正: 将  $=>$  变为  $>=$ 。

- 1.9 a) 抽象。b) 属性。c) 行为。d) 多重。e) 消息。f) 接口。g) 信息隐藏。h) 名词。i) 数据成员、成员函数或方法。j) 对象。

## 练习

- 1.10 将下列项目分为硬件和软件:

- a) CPU
- b) C++ 编译器
- c) ALU
- d) C++ 预处理器
- e) 输入单元
- f) 编译程序

- 1.11 为什么要用机器无关语言而不是用机器相关语言编程?为什么某些程序更适合用机器相关语言编程?

- 1.12 填空:

- a) 计算机的哪个逻辑单元从计算机外部接收计算机使用的信息?\_\_\_\_\_。
- b) 指示计算机解决特定问题的过程称为\_\_\_\_\_。
- c) 哪种计算机语言用机器语言指令的英文缩写?\_\_\_\_\_。
- d) 计算机的哪个逻辑单元将计算机处理过的信息发送到各个设备,并在计算机外部使用?\_\_\_\_\_。
- e) 计算机的哪个逻辑单元保存信息?\_\_\_\_\_。
- f) 计算机的哪个逻辑单元进行计算?\_\_\_\_\_。
- g) 计算机的哪个逻辑单元进行逻辑判断?\_\_\_\_\_。
- h) 最适合程序员方便快捷地编写程序的计算机语言是\_\_\_\_\_。
- i) 计算机能直接理解的惟一语言是该计算机的\_\_\_\_\_。
- j) 计算机的哪个逻辑单元负责协调所有其他逻辑单元的活动?\_\_\_\_\_。

- 1.13 简介下列对象的含义:

- a) cin
- b) cout
- c) cerr

- 1.14 为什么人们特别关心面向对象编程和 C++?

- 1.15 填空:

- a) \_\_\_\_\_ 只是用来说明程序和提高程序可读性。
- b) 在屏幕上打印信息的对象是\_\_\_\_\_。
- c) 进行判断的 C++ 语句是\_\_\_\_\_。
- d) 通常由 \_\_\_\_\_ 语句完成计算。
- e) \_\_\_\_\_ 对象从键盘输入数值。

1.16 编写下列 C++ 语句:

- a) 打印消息 "Enter two numbers".
- b) 将变量 b 和 c 的积赋给变量 a。
- c) 表示程序进行示例工资计算 (即用文本说明程序)。
- d) 从键盘输入整数值并存在整型变量 a、b、c 中。

1.17 判断下列陈述是否正确, 并说明理由。

- a) C++ 运算符从左向右求值。
- b) 下列变量名都是有效变量名: `_under_bar_`、`m928134`、`t5`、`j7`、`her_sales`、`his account_total`、`a`、`b`、`c`、`z`、`z2`。
- c) 语句 `cout << "a = 5;"` 是赋值语句的典型例子。
- d) 不带括号的有效 C++ 算术表达式从左向右求值。
- e) 下列变量名都是无效变量名: `3g`、`87`、`67h2`、`h22`、`2h`。

1.18 填空:

- a) 与乘法的优先级相同的运算符有哪些?\_\_\_\_\_。
- b) 嵌套括号时, 算术表达式中的哪组括号首先求值?\_\_\_\_\_。
- c) 计算机内存中在程序执行期间的不同时间可以包含不同值的地址称为\_\_\_\_\_。

1.19 执行下列 C++ 语句时打印什么内容 (如果有), 如果没有, 回答 "无"。假设  $x = 2$  和  $y = 3$ 。

- a) `cout << x;`
- b) `cout << x + x;`
- c) `cout << "x=";`
- d) `cout << "x = " << x;`
- e) `cout << x + y << " = " << y + x;`
- f) `z = x + y;`
- g) `cin >> x >> y;`
- h) `// cout << "x + y = " << x + y;`
- i) `cout << "\n";`

1.20 下列 C++ 语句哪个包含数值被删除的变量?

- a) `cin >> b >> c >> d >> e >> f;`
- b) `p = i + j + k + 7;`
- c) `cout << "variables whose values are destroyed";`
- d) `coud << "a = 5";`

1.21 对代数方程  $y = ax^3 + 7$ , 下列 C++ 语句哪些是正确的?

- a) `y = a * x * x * x + 7;`
- b) `y = a * x * x * (x + 7);`
- c) `y = (a * x) * x * (x + 7);`
- d) `y = (a * x) * x * x + 7;`
- e) `y = a * (x * x * x) + 7;`
- f) `y = a * x * (x * x + 7);`

1.22 设置下列 C++ 语句中运算符的求值顺序, 并显示执行每个语句之后的 x 值。

- a) `x = 7 + 3 * 6 / 2 - 1;`
- b) `x = 2 % 2 + 2 * 2 - 2 / 2;`

- c)  $x = (3 * 9 * (3 + (9 * 3 / (3))))$ ;
- 1.23 编写程序, 要求用户输入两个数, 从用户取得这两个数, 并打印这两个数的和、积、差、商。
- 1.24 编写程序, 在同一行打印数字 1 到 4, 每两个数之间用一个空格分开。用下列方法编写:
- 用一个输出语句和一个流插入运算符。
  - 用一个输出语句和 4 个流插入运算符。
  - 用 4 个输出语句。
- 1.25 编写程序, 要求用户输入两个数, 从用户取得这两个数, 并打印较大的数加 “is larger”。如果两个数相等, 则打印消息 “These numbers are equal”。
- 1.26 编写程序, 从键盘输入三个值, 并打印其和、平均数、积、最小值和最大值。屏幕对话如下所示:

```
Input three different integers:13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

- 1.27 编写一个程序, 读取圆的半径, 打印圆的直径、周长和面积。 $\pi$  用常量值 3.14159。在输出中进行这些计算 (说明: 本章只介绍了整型常量和整数。第 3 章将介绍浮点数, 即带小数点的值)。
- 1.28 编写一个程序, 打印下图所示的矩形、椭圆、箭头和菱形:

```
*****      ***      *      *
*          *      *      *      ***      * *
*          *      *      *      *      * *
*          *      *      *      *      * *
*          *      *      *      *      *
*
*          *      *      *      *      *      * *
*          *      *      *      *      *      * *
*          *      *      *      *      *      * *
*****      ***      *      *
```

- 1.29 下列代码打印什么结果?

```
cout << " * \n * * \n***\n*****\n*****\n";
```

- 1.30 编写一个程序, 读取 5 个整数并确定和打印其中的最大值。只用本章介绍的编程技术。
- 1.31 编写一个程序, 读取一个整数并确定和打印其为奇数或偶数。提示: 用求模运算符, 偶数是 2 的倍数, 偶数除以 2 的余数为 0。
- 1.32 编写一个程序, 读取两个整数并确定和打印第一个数是否为第二个数的倍数。提示: 用求模运算符。
- 1.33 用八条输出语句显示下列棋盘图案, 然后用更少的语句显示这个棋盘图案。

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```



- 1.34 区分致命错误与非致命错误。为什么希望遇到致命错误而不是非致命错误。
- 1.35 这是个附加题。本章介绍了整数和int类型。C++还可以表示大写字母、小写字母和各种特殊字符。C++内部用小整数表示不同字符。计算机所用字符集及对应的整数表示称为计算机字符集。打印字符时，只需将其放在单引号中，如下所示：

```
cout << 'A';
```

只要在前面加上(int)(称为强制类型转换，详见第2章)，可以打印对应一个字符的整数：

```
cout << (int) 'A';
```

执行上述语句时，它打印数值65(在使用ASCII字符集的系统上)。编写一个程序，打印大写字母、小写字母和各种特殊字符的对应整数。至少确定下列字符和空格符的对应整数：A B C a b c 0 1 2 \$ \* + /。

- 1.36 编写一个程序，读取五位数，将各个位分开，并在打印时在位与位之间增加三个空格。提示：用整除和求模运算符。例如，用户输入42339时，程序打印：

```
4      2      3      3      9
```

- 1.37 用本章学习的方法，编写一个程序，计算0到10的平方和立方，并用制表符打印下列表格值：

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

- 1.38 简述下列“有关对象的思考”问题：
- 为什么本书不是先深入介绍结构化编程之后再深入介绍面向对象编程？
  - 面向对象设计过程的典型步骤如何(见正文)？
  - 人类如何体现多重继承？
  - 人与人之间发送什么消息？
  - 对象在定义良好的接口之间发送消息。车载收音机(对象)与用户(人对象)之间有什么接口？
- 1.39 也许你戴了世界上最常见的对象——手表。请谈谈手表有哪些下列项目：对象、属性、行为、类、继承(例如闹钟)、抽象、模型、消息、封装、接口、信息隐藏、数据成员、成员函数。

## 第2章 控制结构

### 教学目标

- 了解基本问题的解决方法
- 通过自上而下、逐步完善的过程开发算法
- 用 if、if/else 和 switch 选择结构选择操作
- 用 while、do/while 和 for 重复结构重复执行程序语句
- 了解计数器控制重复与标记控制重复
- 使用自增、自减、赋值和逻辑运算符
- 使用 break 和 continue 程序控制语句

### 2.1 简介

编写解决特定问题的程序之前，首先要彻底了解问题并认真计划解决问题的方法。编写程序时，还要了解可用的基本组件和采用实践证明的程序结构原则。本章将讨论结构化编程的理论和原理的所有问题。这里介绍的技术适用于大多数高级语言，包括 C++。第6章在介绍 C++ 面向对象编程时，将会介绍如何用第2章介绍的控制结构帮助建立和操作对象。

### 2.2 算法

任何计算问题都可以通过按特定顺序执行一系列操作而完成。解决问题的过程 (procedure) 称为算法 (algorithm)，包括：

1. 执行的操作 (action)
2. 执行操作的顺序 (order)

下例演示正确指定执行操作的顺序是多么重要：

考虑每个人早晨起床到上班的“朝阳算法”：(1) 起床，(2) 脱睡衣，(3) 洗澡，(4) 穿衣，(5) 吃早饭，(6) 搭车上班。

总裁可以按这个顺序，从容不迫地来到办公室。假设把顺序稍作调换：(1) 起床，(2) 脱睡衣，(3) 穿衣，(4) 洗澡，(5) 吃早饭，(6) 搭车上班。

如果这样，总裁就得带着肥皂水来上班。指定计算机程序执行语句的顺序称为程序控制 (program control)，本章介绍 C++ 程序的控制功能。

## 2.3 伪代码

伪代码 (pseudocode) 是人为的非正式语言, 帮助程序员开发算法。这里介绍的伪代码在开发的算法转换为结构化 C++ 程序时特别有用。伪代码类似于日常英语, 方便而且容易掌握, 但不是实际计算机编程语言。

伪代码程序并不在计算机上实际执行, 而是帮助程序员先“构思”程序, 再用 C++ 之类的实际计算机编程语言编写。本章介绍几个如何在开发结构化 C++ 程序时有效利用伪代码的例子。

我们介绍的伪代码完全由字符构成, 程序员可以用一个编辑器程序方便地输入伪代码程序, 计算机可以在需要时显示伪代码程序。认真构思的伪代码程序可以方便地变为对应的 C++ 程序。很多情况下, 只要将伪代码语句转换成对应的 C++ 语句即可。

伪代码只包含执行语句, 将伪代码程序变为对应的 C++ 程序时, 这些语句可以运行。声明语句不是执行语句。例如, 下列声明:

```
int i;
```

只是告诉编译器, 变量 *i* 的类型是整型, 指示编译器在内存中为这个变量保留内存空间。但这个声明并在执行程序时不做任何操作 (如输入、输出或计算)。有些程序员在伪代码程序开头列出变量及其简要说明。

## 2.4 控制结构

通常, 程序中的语句按编写的顺序一条一条地执行, 称为顺序执行 (sequential execution)。程序员可以用稍后要介绍的不同 C++ 语句指定下一个执行的语句不是紧邻其后的语句, 这种技术称为控制转移 (transfer of control)。

20 世纪 60 年代, 人们发现, 软件开发小组遇到的许多困难都是由于控制转移造成的。goto 语句使程序员可以在程序中任意指定控制转移目标, 因此人们提出结构化编程就是为了清除 goto 语句。

Bohm 和 Jacopini 的研究表明, 不用 goto 语句也能编写程序。困难在于程序员要养成不用 goto 语句的习惯。直到 20 世纪 70 年代, 程序员才开始认真考虑结构化编程, 结果使软件开发小组的开发时间缩短、系统能够及时交付运行并在预算之内完成软件项目。这些成功的关键是, 结构化编程更清晰、更易调试与修改并且不容易出错。

Bohm 和 Jacopini 的研究表明, 所有程序都可以只用三种控制结构 (control structure) 即顺序结构 (sequence structure)、选择结构 (selection structure) 和重复结构 (repetition structure)。顺序结构是 C++ 内置的, 除非另外指定, 计算机总是按编写的顺序一条一条地执行。图 2.1 的流程图 (flowchart) 演示了典型的顺序结构, 按顺序进行两次计算。

流程图是算法或部分算法的图形表示。流程图用一些专用符号绘制, 如长方形、菱形、椭圆和小圆, 这些符号用箭头连接, 称为流程。

和伪代码一样, 流程图也用于开发和表示算法, 但伪代码更受欢迎。流程图能清楚地表示控制结构如何操作, 本书用流程图表示控制结构如何操作。

考虑图 2.1 所示的流程图。我们用矩形框 (或称为执行框) 表示各种操作, 包括计算、输入和输出操作。图中的流程表示进行操作的顺序, 首先将 grade 加进 total, 然后将 counter 加 1。C++ 允

序结构中有多个操作，稍后可以看出，可以放一个操作的地方，也就可以放几个顺序操作。

绘制表示完整算法的流程图时，椭圆框加上其中的“Begin”（开始）字样表示流程图开始，椭圆框加上其中的“End”（结束）字样表示流程图结束。只画部分算法时（如图 2.1），省略椭圆框，只用小圆框，也称接头框。

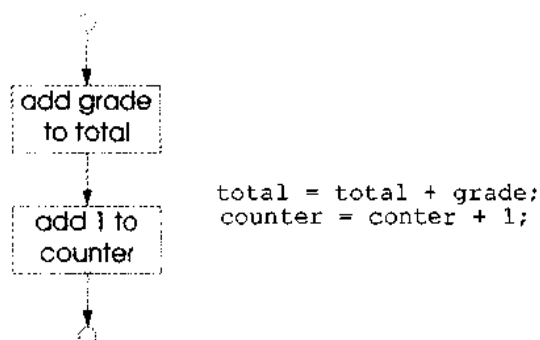


图 2.1 C++ 顺序结构流程图

最重要的流程图符号是菱形框，也称为判断框，表示要进行判断。下一节将介绍菱形框。

C++ 提供三种选择结构，本章将介绍这三种选择结构。if 选择结构在条件为 true 时执行一个操作，在条件为 false 时跳过这个操作。if/else 选择结构在条件为 true 时执行一个操作，在条件为 false 时执行另一个操作。switch 选择结构根据表达式取值不同而选择不同操作。

if 选择结构称为单项选择结构（single-selection structure），选择或忽略一个操作。if/else 选择结构称为双项选择结构（double-selection structure），在两个不同操作中选择。switch 选择结构称为多项选择结构（multiple-selection structure），在多个不同操作中选择。

C++ 提供三种重复结构 while、do/while 和 for。if、else、switch、while、do 和 for 等都是 C++ 关键字（keyword）。这些关键字是该语言保留的，用于实现如 C++ 控制结构等不同特性。关键字不能作为变量名等一些标识符。图 2.2 显示了完整的 C++ 关键字列表。

#### C 与 C++ 关键字

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

#### C++ 特有的关键字

asm	bool	catch	class	const_cast
delete	dynamic_cast	explicit	false	friend
inline	mutable	namespace	new	operator
private	protected	public	reinterpret_cast	
static_cast	template	this	throw	true
try	typeid	typename	using	virtual
wchar_t				

图 2.2 C++ 关键字

### 常见编程错误 2.1

用关键字作为标识符属于语法错误。

C++ 只有七种控制结构：顺序结构、三种选择结构和三种重复结构。每个 C++ 程序都是根据程序所需的算法组合这七种控制结构。从图 2.1 中的顺序结构可以看出，每个控制结构的流程图使用两个小圆框，一个在控制结构入口点，一个在控制结构出口点。这种单入/单出控制结构（single-entry/single-exit control structure）使程序容易建立，只要将一个控制结构的出口与另一个控制结构的入口连接，即可组成程序。这点很像小孩子堆积木，因此称为控制结构堆栈（control-structure stacking），还有另一种控制结构连接方法，称为控制结构嵌套（control-structure nesting）。

### 软件工程视点 2.1

任何 C++ 程序可以用这七种控制结构（顺序、if、if/else、switch、while、do/while 和 for）并用两种方式（控制结构堆栈和控制结构嵌套）组合而成。

## 2.5 if 选择结构

选择结构在不同操作之间选择。例如，假设考试成绩 60 分算及格，则下列伪代码：

```
if student's grade is greater than or equal to 60
    Print "Passed"
```

确定“学生成绩大于或等于 60 分”是 true 或 false，如果是 true，则该生及格，打印“Passed”字样，并顺序“执行”下一个伪代码语句（记住，伪代码不是真正的编程语言）。如果条件为 false，则忽略打印语句，并顺序“执行”下一个伪代码语句。注意这个选择结构第二行的缩排，这种缩排是可选的，但值得提倡，因为它能体现结构化程序的内部结构。将伪代码变成 C++ 代码时，C++ 编译器忽略空格、制表符、换行符等用于缩排和垂直分隔的空白字符。

### 编程技巧 2.1

在整个程序中坚持用合理缩排规则能大大提高程序可读性。建议用固定制表长度即 1/4 英寸或三个空格的缩排量。

上述伪代码的 If 语句可以写成如下 C++ 语句：

```
if (grade >= 60)
    cout << "Passed";
```

注意 C++ 代码与伪代码密切对应，这是伪代码的一个属性，使得其成为有用的程序开发工具。

### 编程技巧 2.2

伪代码常用于程序设计期间“构思”程序，然后再将伪代码程序转换为 C++ 程序。

图 2.3 的流程图演示了单项选择 if 结构。这个流程图包含流程图最重要的菱形框，也称判断框，表示要进行判断。判断框包含一个表达式（如条件），可取 true 或 false 值。判断框产生两条流程，一条指向表达式为 true 时的走向，一条指向表达式为 false 时的走向。第 1 章曾介绍过，可以根据包含关系或相等运算符的条件作出判断。实际上，可以针对任何表达式作出判断，如果表达式求值为 0，则当作 false，如果表达式求值为非 0，则当作 true。C++ 草案标准提供 bool 数据类型，表示 true 和 false。关键字 true 和 false 表示 bool 数据类型的值。

注意, if 结构也是单入/单出结构。稍后将会介绍, 其余控制结构的流程图 (除了小圆框和流程之外) 也只能包含表示所要操作的矩形框和表示所要判断的菱形框。这是我们强调的操作/判断编程模型 (action/decision model of programming)。

可以想像有七个框, 各包含七种控制结构中的一种控制结构, 这些控制结构是空的, 矩形框和菱形框中什么也没有。程序员的任务就是根据算法需要用堆栈和嵌套两种方法组合这几种控制结构, 然后在这些框中填入算法所要的操作和判断, 从而生成程序。下面介绍编写操作和判断的各种方式。

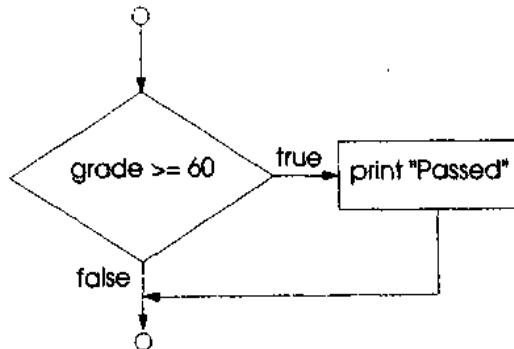


图 2.3 单项选择 if 结构流程图

## 2.6 if/else 选择结构

if 选择结构只在条件为 true 时采取操作, 条件为 false 时则忽略这个操作。利用 if/else 选择结构则可以在条件为 true 时和条件为 false 时采取不同操作。例如, 下列伪代码:

```
If student's grade is greater than or equal to 60
    Print "Passed"
else
    Print "Failed"
```

在学生成绩大于或等于 60 时打印 “Passed”, 否则打印 “Failed”。打印之后, 都 “执行” 下一条伪代码语句。注意 else 的语句体也缩排。

### 编程技巧 2.3

if/else 选择结构的两个语句体都缩排。

选择的缩排规则应当在整个程序中认真贯彻执行。不按统一缩排规则编写的程序很难阅读。

### 编程技巧 2.4

如果有多层缩排, 则每一层应缩排相同的空间量。

上述伪代码 if/else 结构可以写成如下的 C++ 代码:

```
if ( grade >= 60 )
    cout << "Passed";
else
    cout << "Failed";
```

图 2.4 的流程图很好地演示了 if/else 结构的控制流程。注意，这个流程图（除了小圆框和流程之外）也只能包含表示所要操作的矩形框和表示所要判断的菱形框。这里我们继续强调操作/判断模型计算，假设框中包含建立 C++ 程序所需的空白双向选择结构。程序员的任务就是根据算法需要用堆栈和嵌套两种方法组合各种控制结构，然后在这些框中填入算法所要的操作和判断，从而生成程序。

C++ 提供条件运算符（?:），与 if/else 结构密切相关。条件运算符是 C++ 中惟一的三元运算符（ternary operator），即取三个操作数的运算符。操作数和条件运算符一起形成条件表达式（conditional expression）。第一个操作数是条件，第二个操作数是条件为 true 时整个条件表达式的值，第三个操作数是条件为 false 时整个条件表达式的值。例如，下列输出语句：

```
cout << ( grade >= 60? "Passed" : "Failed");
```

包含的条件表达式在 `grade >= 60` 取值为 true 时，求值为字符串 “Passed”；在 `grade >= 60` 取值为 false 时，求值为字符串 “Failed”。这样，带条件表达式的语句实际上与上述 if/else 语句相同。可以看出，条件运算符的优先级较低，因此上述表达式中的括号是必需的。

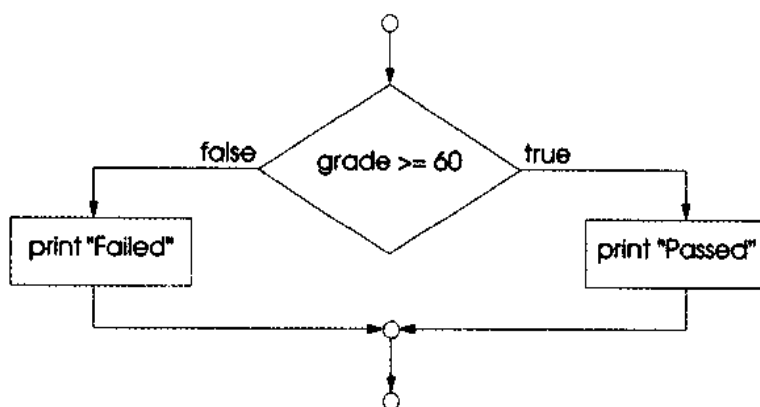


图 2.4 if/else 双向选择结构流程图

条件表达式的值也可以是要执行的操作。例如，下列条件表达式：

```
grade >= 60 ? cout << "Passed" : cout << "Failed";
```

表示如果 `grade` 大于或等于 60，则执行 `cout << "Passed"`，否则执行 `cout << "Failed"`。这与前面的 if/else 结构也是相似的。条件运算符可以在一些无法使用 if/else 语句的情况中使用。

嵌套 if/else 结构（nested if/else structure）测试多个选择，将一个 if/else 选择放在另一个 if/else 选择中。例如，下列伪代码语句在考试成绩大于或等于 90 分时打印 A，在 80 到 89 分之间时打印 B，在 70 到 79 分之间时打印 C，在 60 到 69 分之间时打印 D，否则打印 F。

```
If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
```

```
else
    If student's grade is greater than or equal to 60
        Print "D"
else
    Print "F"
```

这个伪代码对应下列 C++ 代码：

```
if ( grade >= 90 )
    cout << "A";
else
    if ( grade >= 80 )
        cout << "B";
    else
        if ( grade >= 70 )
            cout << "C";
        else
            if ( grade >= 60 )
                cout << "D";
            else
                cout << "F";
```

如果考试成绩大于或等于 90 分，则前 4 个条件都为 true，但只执行第一个测试之后的 cout 语句。执行这个 cout 语句之后，跳过外层 if/else 语句的 else 部分。许多 C++ 程序员喜欢将上述 if 结构写成：

```
if ( grade >= 90 )
    cout << "A";
else if( grade >= 80 )
    cout << "B";
else if( grade >= 70 )
    cout << "C";
else if( grade >= 60 )
    cout << "D";
else
    cout << "F";
```

两种形式是等价的，后者更常用，可以避免深层缩排使代码移到右端。深层缩排会使一行的空间太小，不长的行也要断行，从而影响可读性。

#### 性能提示 2.1

嵌套 if/else 结构比一系列单项选择 if 结构运行速度快得多，因为它能在满足其中一个条件之后即退出。

#### 性能提示 2.2

在嵌套 if/else 结构中，测试条件中 true 可能性较大的应放在嵌套 if/else 结构开头，从而使嵌套 if/else 结构运行更快，比测试不常发生的情况能更早退出。

if 选择结构体中只能有一条语句。要在 if 选择结构体中包括多条语句，就要把这些语句放在花括号 ( {} ) 中。放在花括号中的一组语句称为复合语句 ( compound statement )。

#### 软件工程视点 2.2

复合语句可以放在程序中出现单句语句的任何地方。

下例在 if/else 结构的 else 部分包括复合语句：



```
if ( grade >= 60 )
    cout << "Passed.\n";
else {
    cout << "Failed.\n";
    cout << "You must take this course again.\n";
}
```

如果 grade 小于 60, 则程序执行 else 程序体中的两条语句并打印:

```
Failed.
You must take this course again.
```

注意 else 从句中的两条语句放在花括号中。这些花括号很重要, 如果没有这些花括号, 则下列语句:

```
cout << "You must take this course again.\n";
```

在 if 语句 else 部分之外, 不管成绩是否小于 60 都执行。

#### 常见编程错误 2.2

忽略复合语句中的一个或两个花括号可能在程序中生成语法错误或逻辑错误。

#### 编程技巧 2.5

总是在 if/else 结构 ( 和任何控制结构 ) 中放上花括号, 可以避免不慎疏忽, 特别是后面要在 if 或 else 语句中增加语句时。

语法错误 ( syntax error ) 是由编译器捕获的, 逻辑错误 ( logic error ) 在执行时体现。致命逻辑错误 ( fatal logic error ) 使程序失败和提前终止, 而非致命逻辑错误 ( nonfatal logic error ) 则让程序继续执行, 只是产生错误结果。

#### 软件工程视点 2.3

复合语句可以放在程序中出现单句语句的任何地方, 也可以根本不放语句, 即放上空语句。空语句就是在正常语句出现的地方放一个分号 ( ; )。

#### 常见编程错误 2.3

在 if 结构条件后面放上分号会造成单项选择 if 结构的逻辑错误和双项选择 if 结构的语法错误 ( 如果 if 部分包含实际语句体 )。

#### 编程技巧 2.6

有些程序员喜欢在花括号中输入各个语句之前先输入复合语句的开始花括号和结束花括号, 这样可以避免丢失一个或两个花括号。

本节介绍了复合语句的符号。复合语句可以包含声明 ( 例如, 和 main 程序体中一样 ), 如果这样, 则这个复合语句称为块 ( block )。块中的声明通常放在块中任何操作语句之前, 但也可以和操作语句相混和。第 3 章将介绍块的法, 在此之前, 读者应避免使用块 ( 除了作为 main 程序体 )。

## 2.7 while 重复结构

重复结构 ( repetition structure ) 使程序员可以指定一定条件下可以重复的操作。下列伪代码语句:

```
While there are more items on my shopping list  
Purchase next item and cross it off my list
```

描述购物过程中发生的重复。条件“there are more items on my shopping list”（购物清单中还有更多项目）可真可假。如果条件为 true，则执行操作“Purchase next item and cross it off my list”（购买下一个项目并将其从清单中划去）。如果条件仍然为 true，则这个操作重复执行。while 重复结构中的语句构成 while 的结构体，该结构体可以是单句或复合句。最终，条件会变为 false（购买清单中最后一个项目并将其从清单中划去时），这时重复终止，执行重复结构之后的第一条伪代码语句。

#### 常见编程错误 2.4

如果不在 while 结构中提供最终导致 while 条件变为 false 的操作，则会造成无限循环（infinite loop）错误，重复结构永不终止。

#### 常见编程错误 2.5

将关键字 while 的拼写变为“While”是个语法错误，因为 C++ 是区分大小写的语言。while、if 和 else 等所有 C++ 保留关键字只能包含小写字母。

作为实际 while 的例子，假设程序要寻找 2 的第一个大于 1000 的指数值。假设整数变量 product 初始化为 2，执行下列 while 重复结构之后，product 即会包含所要值：

```
int product = 2;  
while ( product <= 1000 )  
    product = 2 * product;
```

图 2.5 的流程图演示了对应于上述 while 重复结构的 while 结构控制流程。注意，流程图（除了小圆框和流程之外）也只能包含表示所需操作的矩形框和表示所需判断的菱形框。这是我们强调的操作/判断编程模型。程序员的任务就是根据算法需要用堆栈和嵌套两种方法组合其他几种控制结构，然后在这些框中填入算法所要的操作和判断，从而生成程序。流程图中清楚地显示了重复。流程从矩形出发，回到判断框中测试，直到判断为 false。然后退出 while 结构，控制转入程序中下一条语句。

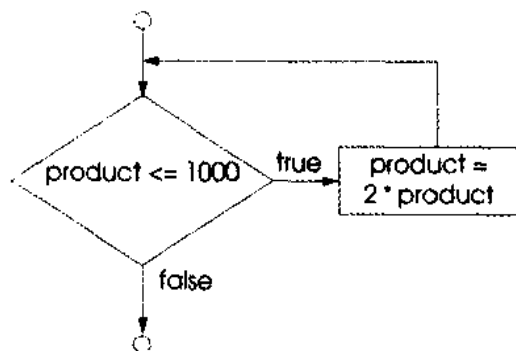


图 2.5 while 重复结构流程图

进入 while 结构时，product 的值为 2。变量 product 重复乘以 2，连续取值 4、8、16、32、64、128、256、512 和 1024。当 product 变为 1024 时，while 结构条件 product <= 1000 变为 false，因此终止重复，product 的最后值为 1024。程序继续执行 while 后面的下一条语句。

## 2.8 构造算法：实例研究 1（计数器控制重复）

要演示如何开发算法，我们要解决几个全班平均成绩的问题。考虑下列问题：

班里有 10 个学生参加测验，可以提供考试成绩（0 到 100 的整数值），以确定全班平均成绩。

全班平均成绩等于全班成绩总和除以班里人数。计算机上解决这个问题的算法是输入每人的成绩，进行平均计算，然后打印结果。

下面用伪代码列出要执行的操作，指定这些操作执行的顺序。我们用计数器控制重复（counter-controlled repetition）一次一个地输入每人的成绩。这种方法用计数器（counter）变量控制一组语句执行的次数。本例中，计数器超过 10 时，停止重复。本节介绍伪代码算法（如图 2.6）和对应程序（如图 2.7）。下节介绍如何开发这个伪代码算法。计数器控制重复通常称为确定重复（definite repetition），因为循环执行之前，已知重复次数。

注意算法中引用了总数（total）和计数器。总数变量用于累计一系列数值的和。计数器变量用于计数，这里计算输入的成绩数。存放总数的变量通常应先初始化为 0 之后再在程序中使用，否则总和会包括总数的内存地址中存放的原有数值。

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

图 2.6 用计数器控制重复解决全班平均成绩问题的伪代码算法

```
1 // Fig. 2.7: fig02_07.cpp
2 // Class average program with counter-controlled repetition
3 #include <iostream.h>
4
5 int main()
6 {
7     int total,           // sum of grades
8         gradeCounter,    // number of grades entered
9         grade,           // one grade
10        average;         // average of grades
11
12    // initialization phase
13    total = 0;            // clear total
14    gradeCounter = 1;     // prepare to loop
15
16    // processing phase
17    while ( gradeCounter <= 10 ) { // loop 10 times
18        cout << "Enter grade: "; // prompt for input
19        cin >> grade;             // input grade
20        total = total + grade;    // add grade to total
21        gradeCounter = gradeCounter + 1; // increment counter
22    }
```

```
23
24         // termination phase
25         average = total / 10;           // integer division
26         cout << "Class average is " << average << endl;
27
28 return 0;    // indicate program ended successfully
29 }
```

**输出结果：**

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

图 2.7 用计数器控制重复解决全班平均成绩问题的 C++ 程序和示例输出

根据使用情况，计数器变量通常应先初始化为 0 或 1（下面会分别举例说明）。未初始化变量会包含垃圾值（“garbage” value），也称为未定义值（undefined value），为该变量保存内存地址中最后存放的值。

**常见编程错误 2.6**

如果不初始化计数器和总和变量，则程序的结果可能不正确，这是一种逻辑错误。

**编程技巧 2.7**

一定要初始化计数器和总和变量。

**编程技巧 2.8**

每个变量在单独一行中声明。

注意程序中的平均计算产生一个整数结果。实际上，本例中的成绩总和是 817，除以 10 时应得到 81.7，是个带小数点的数，下节将介绍如何处理这种值（称为浮点数）。

**常见编程错误 2.7**

在计数器控制循环中，由于循环计数器（每次循环加 1 时）比最大合法值多 1（例如，从 1 算到 10 时为 11），因此在循环之后用计数器值进行计算通常会出现差 1 的错误。

图 2.7 中，如果第 25 行用 gradeCounter 而不是 10 进行计算，则这个程序的输出显示数值 74。

## 2.9 构造算法与自上而下逐步完善：实例研究 2（标记控制重复）

下面将全班平均成绩问题一般化，考虑如下问题：

开发一个计算全班平均成绩的程序，在每次程序运行时处理任意个成绩数。

在第一个全班平均成绩例子中,成绩个数(10)是事先预置的。而本例中,则不知道要输入多少个成绩,程序要处理任意个成绩数。程序怎么确定何时停止输入成绩呢?何时计算和打印全班平均成绩呢?

一种办法是用一个特殊值作为标记值(sentinel value),也称信号值(signal value)、哑值(dummy value)或标志值(flag value),表示数据输入结束(“end of data entry”)。用户输入成绩,直到输入所有合法成绩。然后用户输入一个标记值,表示最后一个成绩已经输入。标记控制重复(sentinel-controlled repetition)也称为不确定重复(indefinite repetition),因为执行循环之前无法事先知道重复次数。

显然,标记值不能与可接受的输入值混淆起来。由于考试成绩通常是非负整数,因此可以用-1作标记值。这样,全班平均成绩程序可以处理95、96、75、74、89和-1之类的输入流。程序计算并打印成绩95、96、75、74和89的全班平均成绩(不计入-1,因为它是标记值)。

#### 常见编程错误 2.8

将选择的标记值与可接受的输入值混淆时会造成逻辑错误。

我们用自上而下逐步完善(top-down, stepwise refinement)的方法开发计算全班平均成绩的程序,这是开发结构化程序的重要方法。我们首先生成上层伪代码表示:

```
Determine the class average for the quiz
```

上层伪代码只是一个语句,表示程序的总体功能。这样,上层等于是程序的完整表达式。但上层通常无法提供编写C++程序所需的足够细节。因此要开始完善过程。我们将上层伪代码分解为一系列的小任务,按其需要完成的顺序列出。这个结果就是下列第一步完善(first refinement):

```
Initialize variables
Input, sum, and count the quiz grades
Calculate and print the class average
```

这里只用了顺序结构,所有步骤按顺序逐步执行。

#### 软件工程视点 2.4

上层伪代码及每一步完善都是算法的完整定义,只是详细程度不同而已。

#### 软件工程视点 2.5

许多程序可以在逻辑上分为三个阶段:初始化阶段将程序变量初始化,处理阶段输入数据值和相应调整程序变量,结束阶段计算和打印最后结果。

上述“软件工程视点”通常是自上而下过程第一步完善的全部工作。要进行下一步完善(即第二步完善, second refinement),我们要指定特定变量,要取得数字的动态和以及计算机处理的数值个数,用一个变量接收每个输入的成绩值,一个变量保存计算平均值。下列伪代码语句:

```
Initialize variables
```

可以细化成:

```
Initialize total to zero
Initialize counter to zero
```

注意, 只有 `total` 和 `counter` 变量要先初始化再使用, `average` 和 `grade` 变量 (分别计算平均值和用户输入) 不需要初始化, 因为它们的值会在计算或输入时重定义。

下列伪代码语句:

```
Input, sum, and count the quiz grades
```

需要用重复结构 (即循环) 连续输入每个成绩。由于我们事先不知道要处理多少个成绩, 因此使用标记控制重复。用户一次一项地输入合法成绩。输入最后一个合法成绩后, 用户输入标记值。程序在每个成绩输入之后测试其是否为标记值, 如果用户输入标记值, 则顺序循环终止。上述伪代码语句的第二步完善如下:

```
Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
Add this grade into the running total
Add one to the grade counter
Input the next grade (possibly the sentinel)
```

注意, 在这个伪代码中, 我们没有在 `while` 结构体中使用花括号, 只是在 `while` 下面将这些语句缩排表示它们属于 `while`。伪代码只是非正式的程序开发辅助工具。

下列伪代码语句可以完善如下:

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"
```

注意我们这里要测试除数为0的可能性, 这是个致命逻辑错误, 如果没有发现, 则会使程序失败 (通常称为爆炸或崩溃)。图 2.8 显示了全班平均成绩问题第二步完善的完整伪代码语句。

#### 常见编程错误 2.9

除数为 0 是个致命逻辑错误。

#### 编程技巧 2.9

进行除法时, 要测试除数为 0 的可能性, 并在程序中进行相应处理 (如打印一个错误消息), 而不是让致命逻辑错误发生。

图 2.6 和图 2.8 的伪代码中增加了一些空行, 使伪代码更易读。空行将程序分成不同阶段。

图 2.8 所示的伪代码算法解决更一般的全班平均成绩问题, 这个算法只进行了第二步完善, 还需要进一步完善。

```
Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
```

```

        Set the average to the total divided by the counter
        Print the average
    else
        Print "No grades were entered"

```

图 2.8 用标记符控制重复解决全班平均成绩问题的伪代码算法

**软件工程视点 2.6**

伪代码算法的细节足以将伪代码变为C++代码时,程序员即可停止自上而下逐步完善的过程,然后就可方便地实现C++程序。

图2.9显示了C++程序和示例执行结果。尽管只输入整数成绩,但结果仍然可能产生带小数点的平均成绩,即实数。`int`类型无法表示实数,程序中引入`float`数据类型处理带小数点的数(也称为浮点数, `floatingpoint number`),并引入特殊的强制类型转换运算符(`cast operator`)处理平均值计算。这些特性将在程序之后详细介绍。

```

1  // Fig. 2.9: fig02_09.cpp
2  // Class average program with sentinel-controlled repetition.
3  #include <iostream.h>
4  #include <iomanip.h>
5
6  int main()
7  {
8      int total,          // sum of grades
9      gradeCounter,      // number of grades entered
10     grade;              // one grade
11     float average;      // number with decimal point for average
12
13     // initialization phase
14     total = 0;
15     gradeCounter = 0;
16
17     // processing phase
18     cout << "Enter grade, -1 to end: ";
19     cin >> grade;
20
21     while ( grade != -1 ) {
22         total = total + grade;
23         gradeCounter = gradeCounter + 1;
24         cout << "Enter grade, -1 to end: ";
25         cin >> grade;
26     }
27
28     // termination phase
29     if ( gradeCounter != 0 ) {
30         average = static_cast< float >( total ) / gradeCounter;
31         cout << "Class average is " << setprecision( 2 )
32              << setiosflags( ios::fixed | ios::showpoint )
33              << average << endl;
34     }
35     else
36         cout << "No grades were entered" << endl;
37
38     return 0;    // indicate program ended successfully

```

```
39 }
```

**输出结果:**

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

图 2.9 用标记符控制重复解决全班平均成绩问题的 C++ 程序和示例执行结果

注意图 2.9 中 while 循环中的复合语句。如果没有花括号，则循环体中的最后三条语句会放到循环以外，使计算机错误地理解如下代码：

```
while ( grade != -1 )
    total = total + grade;
gradeCounter = gradeCounter + 1;
cout << "Enter grade, -1 to end:";
cin >> grade;
```

如果用户输入的第一个成绩不是 -1，则会造成无限循环。

注意下列语句：

```
cin >> grade;
```

前面用一个输出语句提示用户输入。

#### 编程技巧 2.10

提示用户进行每个键盘输入。提示应表示输入形式和任何特殊输入值(如用户终止循环时输入的标记值)。

#### 编程技巧 2.11

在标记控制循环中，提示请求输入数据项目时应显式指定标记值是什么值。

平均值并不一定总是整数值，而常常是包含小数的值，如 7.2 或 ~93.5。这些值称为浮点数，用数据类型 float 表示。变量 average 声明为数据类型 float，以获得计算机结果中的小数。但 total/gradeCounter 的计算结果是整数，因为 total 和 gradeCounter 都是整数变量。两个整数相除是整除 (integer division)，小数部分丢失 (即截尾, truncated)。由于先要进行计算，因此小数部分在将结果赋给 average 之前已经丢失。要用整数值进行浮点数计算，就要先生成用于计算的临时浮点数值。C++ 提供了一元强制类型转换运算符 (unary cast operator)。下列语句：

```
average = static_cast< float >(total) / gradeCounter;
```

包括一元强制类型转换运算符 static\_cast<float>(), 生成用于计算的临时浮点数值 (total)。这样使用强制类型转换运算符称为显式类型转换 (explicit conversion)。total 中存放的值还是整数，而计算时则用浮点数值 (total 的临时 float 版本) 除以整数 gradeCounter。



C++编译器只能对操作数的数据类型一致的表达式求值。要保证操作数的数据类型一致,编译器对所选择的操作数进行提升(promotion)操作(也称为隐式类型转换,implicit conversion)。例如,在包含数据类型float和int的表达式中,int操作数提升为float。本例中,gradeCounter提升为float之后进行计算,将浮点数除法得到的结果赋给average。本章稍后将介绍所有标准数据类型及其提升顺序。任何数据类型都可用强制类型转换运算符,static\_cast运算符由关键字static\_cast加尖括号(<>)中的数据类型名组成。强制类型转换运算符是个一元运算符(unary operator),即只有一个操作数的运算符。第1章曾介绍过二元算术运算符。C++也支持一元正(+)、负(-)运算符,程序员可以编写-7、+5之类的表达式。强制类型转换运算符从右向左结合,其优先级高于正(+)、负(-)运算符等其他一元运算符,该优先级高于运算符\*、/和%,但低于括号的优先级。优先级表中用static\_cast<type>()表示强制类型转换运算符。

图2.9中的格式化功能将在第11章详细介绍,这里先做一简要介绍。下列输出语句中调用setprecision(2):

```
cout << "Class average is" << setprecision(2)
    << setiosflags(ios::fixed | ios::showpoint)
    << average << endl;
```

表示float变量average打印小数点右边的位数为两位精度(precision),例如92.37,这称为参数化流操纵算子(parameterized stream manipulator)。使用这些调用的程序要包含下列预处理指令:

```
#include <iomanip.h>
```

注意endl是非参数化流操纵算子(nonparameterized stream manipulator),不需要iomanip.h头文件。如果不指定精度,则浮点数值通常输出六位精度(即默认精度,default precision),但稍后也会介绍一个例外。

上述语句中的流操纵算子setiosflags(ios::fixed | ios::showpoint)设置两个输出格式选项ios::fixed和ios::showpoint。垂直条(|)分隔setiosflags调用中的多个选项(垂直条将在第16章详细介绍)。选项ios::fixed使浮点数值以浮点格式(而不是科学计数法,见第11章)输出。即使数值为整数,ios::showpoint选项也会强制打印小数点和尾部0,如88.00。如果不用ios::showpoint选项,则C++将该整数显示为88,不打印小数点和尾部0。程序中使用上述格式时,将打印的值取整,表示小数点位数,但内存中的值保持不变。例如,数值87.945和67.543分别输出为87.95和67.54。

#### 常见编程错误 2.10

如果在使用浮点数时认为其精确地表示了数值,则会得到不正确的结果。浮点数在大多数计算机上都采用近似表示。

#### 编程技巧 2.12

不要比较浮点数值相等和不等性,而要测试差值绝对值是否小于指定的值。

尽管浮点数算不总是100%精确,但其用途很广。例如,我们说正常体温98.6(华氏温度)时,并不需要精确地表示,如果温度计上显示98.6度,实际上可能是98.5999473210643度。这里显示98.6对大多数应用已经足够了。

另一种得到浮点数的方法是通过除法。10除以3得到3.333333……,是无限循环小数。计算机只分配固定空间保存这种值,因此只能保存浮点值的近似值。

## 2.10 构造算法与自上而下逐步完善：实例研究3（嵌套控制结构）

下面介绍另一个问题。这里还是用伪代码和自上而下逐步完善的方法构造算法，然后编写相应的C++程序。我们介绍过按顺序堆叠的控制结构，就像小孩堆积木一样。这里显示C++中控制结构的另一种方法，称为嵌套控制结构。

考虑下列问题：

学校开了一门课，让学生参加房地产经纪人证书考试。去年，几个学生读完这门课并参加了证书考试。学校想知道学生考试情况，请编写一个程序来总结这个结果。已经得到了10个学生的名单，每个姓名后面写1时表示考试通过，写2时表示没有通过。

程序应分析考试结果，如下所示：

1. 输入每个考试成绩（即1或2），每次程序请求另一个考试成绩时，在屏幕上显示消息“Enter result”。
2. 计算每种类型的考试成绩数。
3. 显示总成绩，表示及格人数和不及格人数。
4. 如果超过8个学生及格，则打印消息“Raise tuition”。

认真分析上述问题后，我们做出下列结论：

1. 程序要处理10个考试成绩，用计数器控制循环。
2. 每个考试成绩为数字1或2，每次程序读取考试成绩时，程序要确定成绩是否为数字1或2。我们的算法中测试1，如果不是1，则我们假设其为2（本章末尾的练习会考虑这个假设的结果）。
3. 使用两个计数器，分别计算及格人数和不及格人数。
4. 程序处理所有结果之后，要确定是否有超过8个学生及格。

下面进行自上而下逐步完善的过程。首先是上层的伪代码表示：

```
Analyze exam results and decide if tuition should be raised
```

我们再次强调，顶层是程序的完整表达，但通常要先进行几次完善之后才能将伪代码自然演变成C++程序。我们的第一步完善为：

```
Initialize variables  
Input the ten quiz grades and count passes and failures  
Print a summary of the exam results and decide if tuition should be raised
```

这里虽然有整个程序的完整表达式，但还需要进一步完善。我们要提供特定变量。要用两个计数器分别计算，用一个计数器控制循环过程，用一个变量保存用户输入。伪代码语句：

```
Initialize variables
```

可以细分如下：

```
Initialize passes to zero  
Initialize failures to zero
```

```
Initialize student counter to one
```

注意，这里只初始化计数器和总和。伪代码语句：

```
Input the ten quiz grades and count passes and failures
```

要求循环输入每个考试成绩。我们事先知道共有10个成绩，因此可以用计数器控制循环。在循环中（即嵌套在循环中），用一个双项选择结构确定考试成绩为数字1或2，并递增相应的计数器。上述伪代码语句细化如下：

```
While student counter is less than or equal to ten
    Input the next exam result
    If the student passed
        Add one to passes
    else
        Add one to failures
    Add one to student counter
```

注意这里用空行分开 if/else 控制结构，以提高程序可读性。伪代码语句：

```
Print a summary of the exam results and decide if tuition should be raised
```

可以细化如下：

```
Print the number of passes
Print the number of failures
If more than eight students passed
    Print "Raise tuition"
```

图2.10显示了完整的第2步完善结果。注意这里用空行分开 while 结构，以提高程序可读性。

```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to ten
    Input the next exam result
    If the student passed
        Add one to passes
    else
        Add one to failures
    Add one to student counter

Print the number of passes
Print the number of failures
If more than eight students passed
    Print "Raise tuition"
```

图 2.10 检查考试成绩的伪代码

这个伪代码语句已经足以转换为C++程序。图2.11显示了C++程序及示例的执行结果。注意，我们利用C++的一个特性，可以在声明中进行变量初始化。循环程序可能在每次循环开头要求初始化，这种初始化通常在赋值语句中进行。

```
1 // Fig. 2.11: fig02_11.cpp
2 // Analysis of examination results
3 #include <iostream.h>
4
5 int main()
6 {
7     // initialize variables in declarations
8     int passes = 0,           // number of passes
9         failures = 0,        // number of failures
10        studentCounter = 1,   // student counter
11        result;               // one exam result
12
13    // process 10 students; counter-controlled loop
14    while ( studentCounter <= 10 ) {
15        cout << "Enter result (1=pass,2=fail): ";
16        cin >> result;
17
18        if ( result == 1 )      // if/else nested in while
19            passes = passes + 1;
20        else
21            failures = failures + 1;
22
23        studentCounter = studentCounter + 1;
24    }
25
26    // termination phase
27    cout << "Passed " << passes << endl;
28    cout << "Failed " << failures << endl;
29
30    if ( passes > 8 )
31        cout << "Raise tuition " << endl;
32
33    return 0;    // successful termination
34 }
```

**输出结果:**

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Passed 6
Failed 4
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

图 2.11 检查考试成绩的 C++ 程序及示例执行结果

#### 编程技巧 2.13

在声明中进行变量初始化可以帮助程序员避免数据未初始化问题。

#### 软件工程视点 2.7

经验表明, 计算机问题最难解决的部分是开发解决方案的算法。一旦确定正确算法后, 从算法生成 C++ 程序的过程通常是相当简单的。

#### 软件工程视点 2.8

许多熟练的程序员不必用伪代码之类的程序开发工具即可编写程序。这些程序员认为其最终目标是解决计算机上的问题, 编写伪代码只会延迟最终产品的推出。尽管这种方法在简单和熟悉的问题中能行得通, 但在大型复杂项目中则可能导致严重的错误和延迟。

## 2.11 赋值运算符

C++ 提供了几个赋值运算符可以缩写赋值表达式。例如下列语句:

```
c = c + 3;
```

可以用加法赋值运算符 (addition assignment operator) “+=” 缩写如下:

```
c += 3;
```

+= 运算符将运算符右边表达式的值与运算符左边表达式的值相加, 并将结果存放在运算符左边表达式的值中。下列形式的语句:

```
variable = variable operator expression;
```

其中 operator 为二元运算符 +、-、/ 或 % 之一 (或今后要介绍的其他二元运算符), 均可写成如下形式:

```
variable operator = expression;
```

这样, 赋值语句 `c += 3` 将 3 与 c 相加。图 2.12 显示了算术赋值运算符、使用这些算术赋值运算符的示例表达式和说明。

#### 性能提示 2.3

使用缩写赋值运算符可以使程序员更快地编写程序, 也可以使编译器更快地编译程序。有些编译器在用缩写赋值运算符时能产生运行速度更快的代码。

## 性能提示 2.4

本书介绍的许多性能提示只产生少量改进，读者可能不会太注意。但在多次重复的循环中，少量的改进可能积累成巨大的性能改进。

赋值运算符	示例表达式	说明	赋值
假设: int c = 3, d = 5, e = 4, f = 6, g = 12;			
+=	c += 7	c = c + 7	10 赋值给 c
-=	d -= 4	d = d - 4	1 赋值给 d
*=	e *= 5	e = e * 5	20 赋值给 e
/=	f /= 3	f = f / 3	2 赋值给 f
%=	g %= 9	g = g % 9	3 赋值给 g

图 2.12 算术赋值运算符

## 2.12 自增和自减运算符

C++ 还提供一元自增运算符 (increment operator, ++ ) 和一元自减运算符 (decrement operator), 见图 2.13。如果变量 c 递增 1, 则可以用自增运算符 ++, 而不用表达式 c=c+1 或 c+=1。如果将自增和自减运算符放在变量前面, 则称为前置自增或前置递减运算符 (preincrement 或 predecrement operator)。如果将自增和自减运算符放在变量后面, 则称为后置自增或后置自减运算符 (postincrement 或 postdecrement operator)。前置自增 (前置自减) 运算符使变量加 1 (减 1), 然后在表达式中用变量的新值。后置自增 (后置自减) 运算符在表达式中用变量的当前值, 然后再将变量加 1 (减 1)。

运算符	名称	示例表达式	说明
++	前置自增	++a	将 a 加 1, 然后在 a 出现的表达式中使用新值
++	后置自增	a++	在 a 出现的表达式中使用当前值, 然后将 a 加 1
--	前置自减	--b	将 b 减 1, 然后在 b 出现的表达式中使用新值
--	后置自减	b--	在 b 出现的表达式中使用当前值, 然后将 b 减 1

图 2.13 自增和自减运算符

图 2.14 的程序演示了 ++ 运算符的前置自增与后置自增计算之间的差别, 后置自增变量 c 使其在输出语句中使用之后再递增, 而前置自增变量 c 使其在输出语句中使用之前递增。

```

1 // Fig. 2.14: fig02_14.cpp
2 // Preincrementing and postincrementing
3 #include <iostream.h>
4
5 int main()
6 {
7     int c;
8
9     c = 5;
10    cout << c << endl;           // print 5
11    cout << c++ << endl;         // print 5 then postincrement
12    cout << c << endl << endl; // print
13
14    c = 5;
15    cout << c << endl;           // print 5
16    cout << ++c << endl;         // preincrement then print 6
17    cout << c << endl;           // print 6

```

```
18
19     return 0;                // successful termination
20 }
```

**输出结果：**

```
5
5
6

5

6
6
```

图 2.14 前置自增与后置自增计算之间的差别

程序显示使用 ++ 运算符前后的 c 值，自减运算符的用法类似。

#### 编程技巧 2.14

一元运算符及其操作数之间不能插入空格。

图 2.11 的三个赋值语句：

```
passes = passes + 1;
failures = failures + 1
student = student + 1;
```

可以改写成更简练的赋值运算符形式：

```
passes += 1;
failures += 1;
student += 1;
```

使用前置自增运算符，如下所示：

```
++passes;
++failures;
++student;
```

或使用后置自增运算符，如下所示：

```
passes++;
failures++;
student++;
```

注意，单独一条语句中自增或自减变量时，前置自增与后置自增计算之间的结果一样，前置自减与后置自减计算之间的结果也相同。只有变量出现在大表达式中时，才能体现前置自增与后置自增计算之间的差别（和前置自减与后置自减计算之间的差别）。

目前只用简单变量名作为自增和自减的操作数（稍后会介绍，这些运算符也可以用于左值）。

#### 常见编程错误 2.11

要用非简单变量名表达式（如 ++(x+1)）作为自增和自减运算符的操作数是个语法错误。

图 2.15 显示了前面所介绍的运算符优先级和结合律，从上到下，优先级依次递减。第二栏介绍每一级运算符的结合律，注意条件运算符(?:)、一元运算符自增(++)、自减(--)、正(+)、负(-)、强制类型转换以及赋值运算符(=、+=、-=、\*=、/=和%=)的结合律为从右向左。图 2.15 中所有其他运算符的结合律为从左向右。第三栏是运算符的组名。

运算符	结合律	类型
()	从左向右	括号
++    --    +    -    static_cast<type>()	从右向左	一元
*    /    %	从左向右	乘
+    -	从左向右	加
<<    >>	从左向右	插入/读取
<    <=    >    >=	从左向右	关系
==    !=	从左向右	相等
?:	从右向左	条件
=    +=    -=    *=    /=    %=	从右向左	赋值
,	从左向右	逗号

图 2.15 前面所介绍的运算符优先级和结合律

## 2.13 计数器控制循环的要点

计数器控制循环要求：

1. 控制变量（或循环计数器）的名称（name）。
2. 控制变量的初始值（initial value）。
3. 测试控制变量终值（final value）的条件（即是否继续循环）。
4. 每次循环时控制变量修改的增量或减量（increment 或 decrement）。

考虑图 2.16 所示的简单程序，打印 1 到 10 的数字。声明：

```
int counter = 1;
```

指定控制变量（counter）并声明为整数，在内存中为其保留空间并将初始值设置为 1。需要初始化的声明实际上是可执行语句。在 C++ 中，将需要分配内存的声明称为定义（definition）更准确。

```
1 // Fig. 2.16: fig02_16.cpp
2 // Counter-controlled repetition
3 #include <iostream.h>
4
5 int main()
6 {
7     int counter = 1;           // initialization
8
9     while ( counter <= 10 ) {  // repetition condition
10         cout << counter << endl;
11         ++counter;             // increment
12     }
13
14     return 0;
```



```
15 }
```

**输出结果:**

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

图 2.16 计数器控制循环

counter 的声明和初始化也可以用下列语句完成:

```
int counter;  
counter = 1;
```

声明不是可执行语句,但赋值是可执行语句。我们用两种方法将变量初始化。

下列语句:

```
++counter;
```

在每次循环时将循环计数器的值加1。while结构中的循环条件测试控制变量的值是否小于或等于10 (条件为true的终值)。注意,即使控制变量是10时,这个while结构体仍然执行。控制变量超过10时(即counter变成11时),循环终止。

图 2.16 的程序也可以更加简化,将counter初始化为0并将while结构换成:

```
while (++counter <= 10)  
    cout << counter << endl;
```

这段代码减少了语句,直接在while条件中先增加计数器的值再测试条件。这段代码还消除了while结构体的花括号,因为这时while只包含一条语句。

#### 常见编程错误 2.12

由于浮点值可能是近似值,用浮点变量控制计数循环可能导致不精确的计数器值,使测试的结果不准确。

#### 编程技巧 2.15

用整数值控制计数循环。

#### 编程技巧 2.16

缩排每个控制结构体中的语句。

#### 编程技巧 2.17

在每个控制结构前后加上空行,使其在程序中一目了然。

#### 编程技巧 2.18

嵌套太多会使程序难以理解。一般来说,缩排不宜超过三层。

**编程技巧 2.19**

在每个控制结构前后加上空行,并编排每个控制结构体中的语句使程序产生二维效果,大大增加可读性。

## 2.14 for 重复结构

for 重复结构处理计数器控制循环的所有细节。要演示 for 的功能,可以改写图 2.16 的程序,结果如图 2.17。

执行 for 重复结构时,声明控制变量 counter 并将其初始化为 1。然后检查循环条件 counter <= 10。由于 counter 的初始值为 1,因此条件满足,打印 counter 的值(1)。然后在表达式 counter++ 中递增控制变量 counter,再次进行循环和测试循环条件。由于这时控制变量等于 2,没有超过最后值,因此程序再次执行语句体。这个过程一直继续,直到控制变量 counter 递增到 11,使循环条件的测试失败,重复终止。程序继续执行 for 结构后面的第一条语句(这里是程序末尾的 return 语句)。

```
1 // Fig. 2.17: fig02_17.cpp
2 // Counter-controlled repetition with the for structure
3 #include <iostream.h>
4
5 int main()
6 {
7     // Initialization, repetition condition, and incrementing
8     // are all included in the for structure header.
9
10    for ( int counter = 1; counter <= 10; counter++ )
11        cout << counter << endl;
12
13    return 0;
14 }
```

图 2.17 用 for 结构的计数器控制重复

图 2.18 更进一步研究了图 2.17 中的 for 结构。注意 for 结构指定计数器控制重复所需的每个项目。如果 for 结构体中有多条语句,则应把语句体放在花括号中。

注意图 2.17 用循环条件 counter <= 10。如果循环条件变为 counter < 10,则循环只执行 9 次,这种常见的逻辑错误称为差 1 错误。

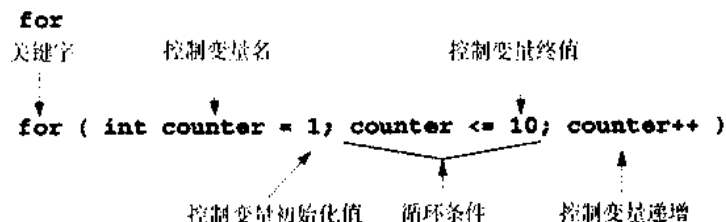


图 2.18 典型 for 首部的组件

**常见编程错误 2.13**

while 或 for 条件中使用不正确的关系运算符和不正确的循环计数器终值会导致差 1 的错误。

**编程技巧 2.20**

在 while 或 for 条件中使用终值和用关系运算符  $\leq$  可以避免差 1 的错误。例如，对打印 1 到 10 的循环，循环条件应为 `counter  $\leq$  10` 而不是 `counter < 10`（导致差 1 错误）或 `counter`（虽然正确）。许多程序员喜欢基于 0 的计数，在循环中重复 10 次，`counter` 初始化为 0，循环条件测试为 `counter < 10`。

for 结构的一般格式如下：

```
for (expression1; expression2; expression3)
    statement
```

其中 `expression1` 初始化循环控制变量的值，`expression2` 是循环条件，`expression3` 递增控制变量。大多数情况下，for 结构可以表示为等价的 while 结构：

```
expression1
while (expression2){
    statement
    expression3;
}
```

惟一的例外将在 2.18 节介绍。

如果 for 结构首部中的 `expression1`（初始化部分）定义控制变量（即控制变量类型在变量名前面指定），则该控制变量只能在 for 结构体中使用，即控制变量值是 for 结构之外所未知的。这种限制控制变量名的用法称为变量的作用域（scope）。变量的作用域定义其在程序中的使用范围。作用域将在第 3 章“函数”中介绍。

**常见编程错误 2.14**

如果 for 结构首部中的初始化部分定义控制变量，则在该结构体之后使用这个控制变量是个语法错误。

**可移植性提示 2.1**

在新的 C++ 草案标准中，for 结构初始化部分声明的控制变量范围与旧式的 C++ 编译器中不同。旧式的 C++ 编译器产生的 C++ 代码在支持新的 C++ 草案标准的编译器中编译时可能遭到破坏。可以用两个编程策略防止这个问题：在每个 for 结构中定义不同名称的控制变量或者在多个 for 结构中定义相同名称的控制变量，并在第一个 for 循环之外和之前定义控制变量。

有时，`expression1` 和 `expression3` 是由逗号分开的表达式列表。这里用逗号作为逗号运算符（comma operator），保证从左向右求值表达式列表。逗号运算符在所有 C++ 运算符中的优先级最低。逗号分隔表达式列表的值和类型是列表中最右边表达式的值和类型。逗号运算符最常用于 for 结构，其主要用途是让程序员使用多个初始化表达式或多个递增表达式。例如，一个 for 结构中可能有多控制变量需要初始化和递增。

**编程技巧 2.21**

只把涉及控制变量的表达式放在 for 结构的初始化和递增部分。其他变量的操作应放在循环之前（如果像初始化语句一样只执行一次）或循环体中（如果对每个循环执行一次，如递增和递减语句）。

for 结构中的三个表达式是可选的。如果省略 `expression2`，则 C++ 假设循环条件为真，从而生成无限循环。如果程序其他地方初始化控制变量，则可以省略 `expression1`。如果 for 语句体中的语

句计算增量或不需要增量,则可以省略 expression3。for 结构中的增量表达式就像是 for 语句体末尾的独立语句。因此,下列表达式:

```
counter = counter + 1
counter += 1
++counter
counter++
```

在 for 结构的递增部分都是等价的。许多程序员喜欢 counter++, 因为递增在执行循环体之后发生,因此,后置自增形式似乎更自然。由于这里递增的变量没有出现在表达式中,因此前置自增与后置自增的效果相同。for 结构首部中的两个分号是必需的。

#### 常见编程错误 2.15

for 结构首部中的两个分号改成逗号会造成语法错误。

#### 常见编程错误 2.16

将分号放在 for 结构首部的右括号后面会使该 for 结构体变为空语句,通常是个逻辑错误。

#### 软件工程视点 2.9

将分号放在紧接着 for 结构首部的后面有时可以生成所谓的延迟循环。这种 for 的循环体是空语句,表示计算空循环的次数。例如,可以用空循环减慢程序速度,以避免其在屏幕上输出太快,无法阅读。

for 结构的初始化、循环条件和递增部分可以用算术表达式。例如,假设 x=2 和 y=10,如果 x 和 y 的值在循环体中不被修改,则下列语句:

```
for ( int j = x; j <= 4* x* y; j += y / x )
```

等于下列语句:

```
for ( int j = 2; j <= 80; j += 5 )
```

for 结构的增量也可能是负数(实际上是递减,循环向下计数)。

如果循环条件最初为 false,则 for 结构体不执行,执行 for 后面的语句。

for 结构中经常打印控制变量或用控制变量进行计算,控制变量常用于控制重复而不在 for 结构体中提及这些控制变量。

#### 编程技巧 2.22

尽管控制变量值可以在 for 循环体中改变,但最好不要这样做,因为这样可能造成一定的逻辑错误。

for 结构的流程图与 while 结构相似。例如,图 2.19 显示了下列 for 语句的流程图:

```
for ( int counter = 1; counter >=10; counter++ )
    cout << counter << endl;
```

从这个流程图可以看出初始化发生一次,并在每次执行结构体语句之后递增。注意,流程图(除了小圆框和流程之外)也只能包含表示操作的矩形框和表示判断的菱形框。这是我们强调的操作/判断编程模型。程序员的任务就是根据算法使用堆栈和嵌套两种方法组合其他几种控制结构,然后在这些框中填入算法所要的操作和判断,从而生成程序。

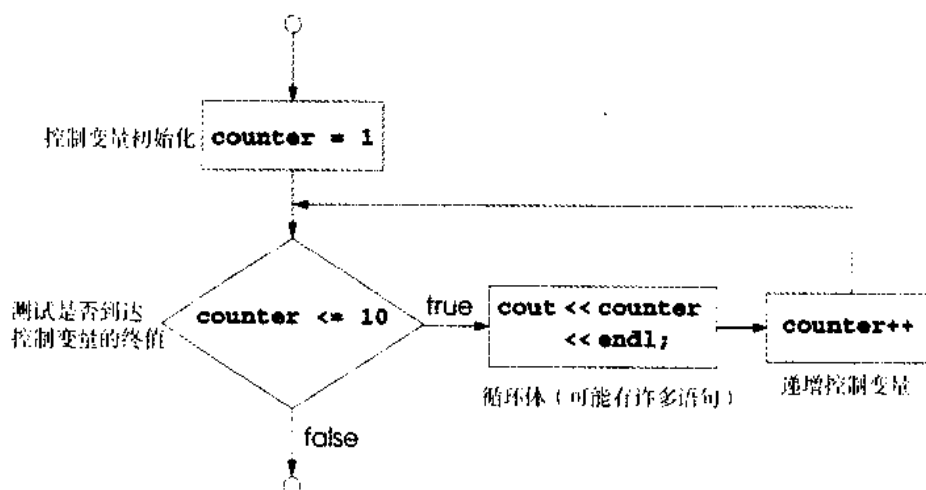


图 2.19 典型 for 语句的流程图

## 2.15 for 结构使用举例

下面的例子显示 for 结构中改变控制变量的方法。在每个例子中，我们都编写相应的 for 结构首部。注意循环中递减控制变量的关系运算符的改变。

a) 将控制变量从 1 变到 100，增量为 1。

```
for (int i = 1; i <= 100; i++)
```

b) 将控制变量从 100 变到 1，增量为 -1。

```
for (int i = 100; i >= 1; i--)
```

### 常见编程错误 2.17

循环向下计数时如果循环条件中不使用正确的关系运算符（如在向下计算到 1 的循环中使用  $i \leq 1$ ）通常是个逻辑错误，会在程序运行时产生错误结果。

c) 控制变量的变化范围为 7 到 77。

```
for (int i = 7; i <= 77; i += 7)
```

d) 控制变量的变化范围为 20 到 2。

```
for (int i = 20; i >= 2; i -= 2)
```

e) 按所示数列改变控制变量值：2、5、8、11、14、17、20。

```
for (int j = 2; j <= 20; j += 3)
```

f) 按所示数列改变控制变量值：99、88、77、66、55、44、33、22、11、0。

```
for (int j = 99; j >= 0; j -= 11)
```

下面两个例子提供 for 结构的简单应用。图 2.20 所示的程序用 for 结构求 2 到 100 的所有整数的总和。

注意图 2.20 中 for 结构体可以用下列逗号运算符合并成 for 首部的右边部分：

```
for (int number = 2;                // initialization
    number <= 100;                  // continuation condition
    sum += number, number += 2)     // total and increment
;
```

初始化 sum=0 也可以合并到 for 的初始化部分。

```
1 // Fig. 2.20: fig02_20.cpp
2 // Summation with for
3 #include <iostream.h>
4
5 int main()
6 {
7     int sum = 0;
8
9     for ( int number = 2; number <= 100; number += 2 )
10         sum += number;
11
12     cout << "Sum is " << sum << endl;
13
14     return 0;
15 }
```

**输出结果：**

sum is 2550

图 2.20 用 for 语句求和

#### 编程技巧 2.23

尽管 for 前面的语句和 for 结构体的语句通常可以合并到 for 的首部中，但最好不要这么做，因为这样会使程序更难阅读。

#### 编程技巧 2.24

尽可能将控制结构首部的长度限制为一行。

下列用 for 结构计算复利。考虑下列问题：

一个人在银行存款 1000.00 美元，每利率为 5%。假设所有利息留在账号中，则计算 10 年间每年年末的金额并打印出来。用下列公式求出金额：

$$a = p(1+r)^n$$

其中：

p 是原存款（本金）

r 是年利率

n 是年数

a 是年末本息

这个问题要用一个循环对 10 年的存款进行计算。解答如图 2.21。

for结构执行循环体10次，将控制变量从1变到10，增量为1。C++中没有指数运算符，因此要用标准库函数pow。函数pow(x, y)计算x的y次方值。函数pow取两个类型为double的参数并返回double值。类型double与float相似，但double类型的变量能存放比float精度更大的数值。C++把常量（如图2.21中的1000.0和.05）当作double类型处理。

```
1 // Fig. 2.21: fig02_21.cpp
2 // Calculating compound interest
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <math.h>
6
7 int main()
8 {
9     double amount,           // amount on deposit
10         principal = 1000.0, // starting principal
11         rate = .05;         // interest rate
12
13     cout << "Year" << setw( 21 )
14         << "Amount on deposit" << endl;
15
16     for ( int year = 1; year <= 10; year++ ) {
17         amount = principal * pow( 1.0 + rate, year );
18         cout << setw( 4 ) << year
19             << setiosflags( ios::fixed | ios::showpoint )
20             << setw( 21 ) << setprecision( 2 )
21             << amount << endl;
22     }
23
24     return 0;
25 }
```

**输出结果：**

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.62
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

图 2.21 用for结构计算复利

这个程序必须包括math.h才能编译。函数pow要求两个double参数，注意year是个整数。math.h文件中的信息告诉编译器将year值转换为临时double类型之后再调用函数。这些信息放在pow的函数原型（function prototype）中。第3章将介绍函数原型并总结pow函数和其他数学库函数。

**常见编程错误 2.18**

程序中使用数学库函数而不包括math.h头文件是个语法错误。

程序中将变量 `amount`、`principal` 和 `rate` 声明为 `double` 类型，这是为了简单起见，因为我们要涉及存款数额的小数部分，要采用数值中允许小数的类型。但是，这可能造成麻烦，下面简单介绍用 `float` 和 `double` 表示数值时可能出现的问题（假设打印时用 `setprecision(2)`）：机器中存放的两个 `float` 类型的值可能是 14.234（打印 14.23）和 18.673（打印 18.67）。这两个值相加时，内部和为 32.907，打印 32.91。结果输出如下：

$$\begin{array}{r} 14.23 \\ + 18.67 \\ \hline 32.91 \end{array}$$

但这个加式的和应为 32.90。

#### 编程技巧 2.25

不要用 `float` 和 `double` 表示一些货币值。浮点数是不精确的，可能导致错误，产生不准确的货币值。练习中将介绍用整数值进行货币计算。注意：C++ 类库可以用于正确地进行货币计算。

输出语句：

```
cout << setw(4) << year
      << setiosflags ( ios::fixed | ios::showpoint )
      << setw( 21 ) << setprecision(2)
      << amount << endl;
```

用参数化流操纵算子 `setw`、`setiosflags` 和 `setprecision` 指定的格式打印变量 `year` 和 `amount` 的值。调用 `setw(4)` 指定下一个值的输出域宽（field width）为 4，即至少用 4 个字符位置打印这个值。如果输出的值宽度少于 4 个字符位，则该值默认在输出域中右对齐（right justified）。如果输出的值宽度大于 4 个字符，则该值将输出域宽扩大到能放下这个值为止。调用 `setiosflags(ios::left)` 可以指定输出的值为左对齐（left justified）。

上述输出中的其余格式表示变量 `amount` 打印成带小数点的定点值（用 `setiosflags ( ios::fixed | ios::showpoint )` 指定），在输出域的 21 个字符位中右对齐（用 `setw(21)` 指定），小数点后面为两位（用 `setprecision(2)` 指定）。我们将在第 11 章介绍 C++ 强大的输入/输出格式功能。

注意计算 `1.0 + rate` 作为 `pow` 函数的参数，包含在 `for` 语句体中。事实上，这个计算产生的结果在每次循环时相同，因此是重复计算（是一种浪费）。

#### 性能提示 2.5

避免把不改变数值的表达式放在循环中。但即使把不改变数值的表达式放在循环中，如今的许多复杂优化编译器也会自动地把这种表达式放到循环之外，产生优化机器语言代码。

#### 性能提示 2.6

许多编译器中有优化特性，可以改进所写的代码，但最好一开始就编写优化的代码。

为了增加趣味性，本章练习中提供了一个 Peter Minuit 问题，演示了复利计算的精彩之处。

## 2.16 switch 多项选择结构

前面介绍了 `if` 单项选择结构和 `if/else` 双项选择结构。有时算法中包含一系列判断，用一个变量或表达式测试每个可能的常量值，并相应采取不同操作。C++ 提供的 `switch` 多项选择结构可以进行这种判断。



switch结构包括一系列case标记和一个可选default情况。图2.22中的程序用switch计算学生考试的每一级人数。

```
1 // Fig. 2.22: fig02_22.cpp
2 // Counting letter grades
3 #include <iostream.h>
4
5 int main()
6 {
7     int grade,          // one grade
8     aCount = 0,        // number of A's
9     bCount = 0,        // number of B's
10    cCount = 0,        // number of C's
11    dCount = 0,        // number of D's
12    fCount = 0;        // number of F's
13
14    cout << "Enter the letter grades." << endl
15         << "Enter the EOF character to end input." << endl;
16
17    while ( ( grade = cin.get() ) != EOF ) {
18
19        switch ( grade ) {          // switch nested in while
20
21            case 'A': // grade was uppercase A
22            case 'a': // or lowercase a
23                ++aCount;
24                break; // necessary to exit switch
25
26            case 'B': // grade was uppercase B
27            case 'b': // or lowercase b
28                ++bCount;
29                break;
30
31            case 'C': // grade was uppercase C
32            case 'c': // or lowercase c
33                ++cCount;
34                break;
35
36            case 'D': // grade was uppercase D
37            case 'd': // or lowercase d
38                ++dCount;
39                break;
40
41            case 'F': // grade was uppercase F
42            case 'f': // or lowercase f
43                ++fCount;
44                break;
45
46            case '\n': // ignore newlines,
47            case '\t': // tabs,
48            case ' ': // and spaces in input
49                break;
50
51            default: // catch all other characters
52                cout << "Incorrect letter grade entered."
```

```

53             << " Enter a new grade." << endl;
54             break; // optional
55         }
56     }
57
58     cout << "\n\nTotals for each letter grade are:"
59         << "\nA: " << aCount
60         << "\nB: " << bCount
61         << "\nC: " << cCount
62         << "\nD: " << dCount
63         << "\nF: " << fCount << endl;
64
65     return 0;
66 }

```

**输出结果:**

```

Enter the letter grades.
Enter the EOF character to end input.
A
B
C
C
A
D
F
C
E
Incorrect letter grade entered.Enter a new grade.
D
A
B

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1

```

图 2.22 使用 switch 举例

程序中，用户输入一个班代表成绩的字母。在 while 首部中：

```
while ( (grade = cin.get()) != EOF)
```

首先执行参数化赋值 (grade = cin.get())。cin.get() 函数从键盘读取一个字符，并将这个字符存放在整型变量 grade 中。cin.get() 中使用的圆点符号将在第 6 章中介绍。字符通常存放在 char 类型的变量中，但是，C++ 的一个重要特性是可以用任何整数数据类型存放字符，因为它们在计算机中表示为 1 个字节的整数。这样，我们根据使用情况，可以把字符当作整数或字符。例如，下列语句：

```
cout << "The character(" << 'a' << ") has the value "
      << static_cast<int> ('a') << endl;
```

打印字符 a 及其整数值如下所示:

```
The character (a) has the value 97
```

整数 97 是计算机中该字符的数字表示。如今许多计算机都使用 ASCII (American Standard Code for Information Interchange, 美国标准信息交换码) 字符集 (character set), 其中 97 表示小写字母 “a”。附录中列出了 ASCII 字符及其十进制值的列表。

赋值语句的整个值为等号左边变量指定的值。这样, 赋值语句 `grade = cin.get()` 的值等于 `cin.get()` 返回的值, 赋给变量 `grade`。

赋值语句可以用于同时初始化多个变量。例如:

```
a = b = c = 0;
```

首先进行赋值 `c = 0` (因为 `=` 运算符从右向左结合), 然后将 `c = 0` 的值赋给变量 `b` (为 0), 最后将 `b = (c = 0)` 的值赋给变量 `a` (也是 0)。程序中, 赋值语句 `grade = cin.get()` 的值与 EOF 值 (表示文件结束的符号) 比较。我们用 EOF (通常取值为 -1) 作为标记值。用户输入一个系统识别的组合键, 表示文件结束, 没有更多要输入的数据。EOF 是 `<iostream.h>` 头文件中定义的符号化整型常量。如果 `grade` 的取值为 EOF, 则程序终止。我们选择将这个程序中的字符表示为 `int`, 因为 EOF 取整数值 (通常取值为 -1)。

#### 可移植性提示 2.2

表示文件结束的组合键与系统有关。

#### 可移植性提示 2.3

测试符号化常量 EOF 而不是测试 -1 能使程序更容易移植。ANSI 标准要求 EOF 取负整数值, 但不一定是 -1。这样, EOF 在不同系统中可能取不同的值。

在 UNIX 和许多其他系统中, 表示文件结束的输入如下:

```
<ctrl-d>
```

即同时按 `ctrl` 键和 `d` 键。而在 DEC 公司的 VAX VMS 或 Microsoft 公司的 MS-DOS 等系统中, 表示文件结束的输入如下:

```
<ctrl-z>
```

用户通过键盘输入成绩。按 `Enter` (或 `Return`) 键时, `cin.get()` 函数一次读取一个字符。如果输入的字符不是文件结束符, 则进入 `switch` 结构。关键字 `switch` 后面是括号中的变量名 `grade`, 称为控制表达式 (controlling expression)。控制表达式的值与每个 `case` 标记比较。假设用户输入成绩 C, 则 C 自动与 `switch` 中的每个 `case` 比较。如果找到匹配的 (`case 'C':`), 则执行该 `case` 语句。对于字母 C 的 `case` 语句中, `cCount` 加 1, 并用 `break` 语句立即退出 `switch` 结构。注意, 与其他控制结构不同的是, 有多个语句的 `case` 不必放在花括号中。

`break` 语句使程序控制转到 `switch` 结构后面的第一条语句。`break` 语句使 `switch` 结构中的其他 `case` 不会一起运行。如果 `switch` 结构中不用 `break` 语句, 则每次结构中发现匹配时, 执行所有余下 `case` 中的语句 (这个特性在几个 `case` 要完成相同操作时有用, 见图 2.22 的程序)。如果找不到匹配, 则执行 `default case` 并打印一个错误消息。

每个 `case` 有一个或几个操作。`switch` 结构与其他结构不同, 多个语句的 `case` 不必放在花括号中。图 2.23 显示了一般的 `switch` 多项选择结构 (每个 `case` 用一个 `break` 语句) 的流程图。

从流程图中可以看出, case 末尾的每个 break 语句使控制立即退出 switch 结构。注意, 流程图(除了小圆框和流程之外)也只能包含矩形框和菱形框。这是我们强调的操作/判断编程模型。程序员的任务就是根据算法需要用堆栈和嵌套两种方法组合其他几种控制结构, 然后填入算法所要的操作和判断, 从而生成程序。嵌套控制结构很常见, 但程序中很少出现嵌套 switch 结构。

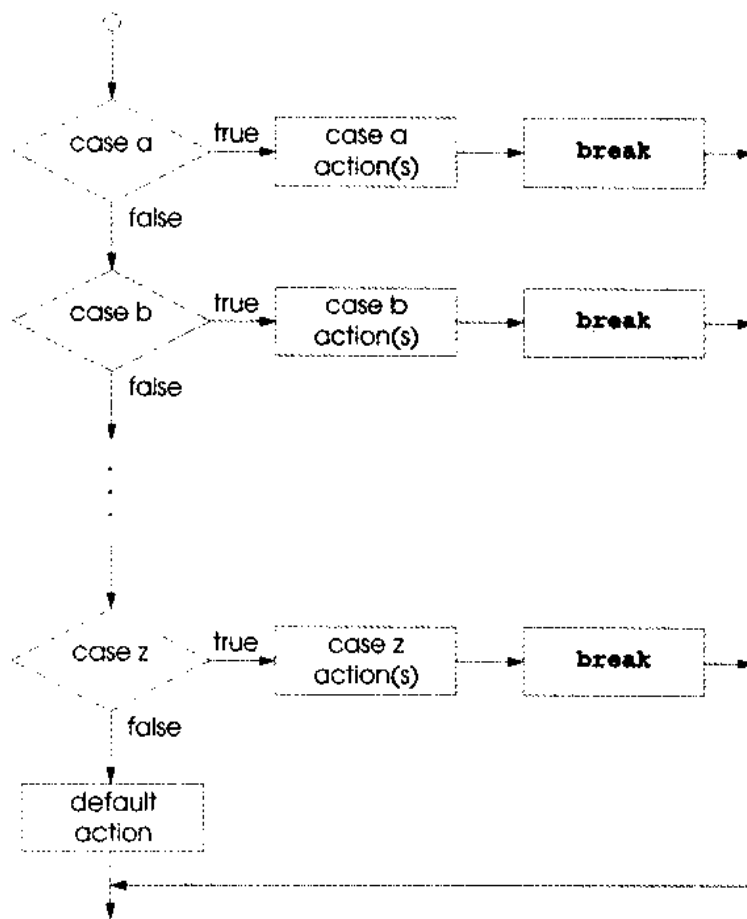


图 2.23 带有 break 语句的 switch 多项选择结构

#### 常见编程错误 2.19

switch 结构中需要 break 语句时而忘记使用 break 语句是个逻辑错误。

#### 常见编程错误 2.20

省略 switch 结构中 case 字样与测试值之间的空格可能造成逻辑错误。例如, 写成 case3: 而不是写成 case 3: 只会生成一个无用标号(详见第 18 章)。这样, switch 结构的控制表达式取值为 3 时, switch 结构不会进行相应操作。

#### 编程技巧 2.26

应在 switch 结构中提供 default case。switch 语句中没有显式测试的 case 在没有 default case 时自动忽略。而包括 default case 则可以提醒程序员需要处理异常条件, 但有时也不需要 default 处理。

#### 编程技巧 2.27

尽管 switch 结构中的 case 语句和 default case 语句可以按任何顺序出现, 但比较好的编程习惯是把 default case 放在最后。

**编程技巧 2.28**

把 default case 放在 switch 结构中的最后不需要 break 语句。有些程序员加上这个 break 以使程序更清晰,并和其他 case 对称。

在图 2.22 中的 switch 结构中,第 46 到第 49 行:

```
case '\n':  
case '\t':  
case ' ':  
    break;
```

使程序跳过换行符、制表符和空白字符。一次读取一个字符可能造成一些问题。要让程序读取字符,就要按键盘上的 Enter 键将字符发送到计算机中,在要处理的字符后面输入换行符。这个换行符通常需要特殊处理,才能使程序正确工作。通过在 switch 结构中包含 case 语句,可以防止在每次输入中遇到换行符、制表符或空格时 default case 打印错误消息。

**常见编程错误 2.21**

一次读取一个字符时不处理输入中遇到的换行符和其他空白字符可能造成逻辑错误。

注意几个标号列在一起时(如图 2.22 中的 case 'D': case 'd':)表示每个 case 发生一组相同的操作。

使用 switch 结构时,记住它只用于测试常量整型表达式(constant integral expression),即求值为一个常量整数值字符常量和整型常量的组合。字符常量表示为单引号中的特定字符(如 'A'),而整型常量表示为整数值。

本书介绍面向对象编程时,会介绍实现 switch 逻辑的更精彩方法。我们使用多态方法生成比使用 switch 逻辑的程序更清晰、更简洁、更易维护和扩展的程序。

C++ 之类可移植语言应有更灵活的数据类型长度,不同应用可能需要不同长度的整数。C++ 提供了几种表示整数的数据类型。每种类型的整数值范围取决于特定的计算机硬件。除了类型 int 和 char 外, C++ 还提供 short (short int 的缩写) 和 long (long int 的缩写) 类型。short 整数的取值范围是  $\pm 32767$ 。对于大多数整数计算,使用 long 类型的整数已经足够。long 整数的取值范围是  $\pm 2147483647$ 。在大多数系统中, int 等价于 short 或 long。int 的取值范围在 short 和 long 的取值范围之间。char 数据类型可以表示计算机字符集中的任何字符, char 数据类型也可以表示小整数。

**可移植性提示 2.4**

由于 int 的长度随系统不同而不同,因此如果要处理超过  $\pm 32767$  的数值,应使用 long 整数,以便在几个不同计算机系统中运行。

**性能提示 2.7**

在内存有限或要求执行速度的面向性能的情况下,可以考虑用较小的整数长度。

**性能提示 2.8**

如果操作程序的机器指令不如自然长度整数那么有效(例如要进行符号扩展),则用较小的整数长度会使程序减慢。

**常见编程错误 2.22**

在 switch 结构中提供相同的 case 标号是个语法错误。

## 2.17 do/while 重复结构

do/while 重复结构与 while 结构相似。在 while 结构中，先在循环开头测试循环条件之后再执行循环体。do/while 重复结构执行循环体之后再测试循环条件，因此，do/while 结构至少执行循环体一次。do/while 结构终止时，继续执行 while 语句后面的语句。注意，如果结构体中只有一条语句，则不必在 do/while 结构中使用花括号。但通常还是加上花括号，避免混淆 while 与 do/while 重复结构。例如：

```
while ( condition )
```

通常当作 while 结构的首部。结构体中只有一条语句的 do/while 结构中不使用花括号时：

```
do
    statement
while ( condition );
```

最后一行 while ( condition ) 可能被误解成 while 结构包含空语句。这样，只有一个语句的 do/while 结构通常写成如下形式：

```
do {
    statement
} while ( condition );
```

### 编程技巧 2.29

即使不需要花括号时，一些程序员也在 do/while 结构中加上花括号，这样可以区分只有一个语句的 do/while 结构与 while 结构。

### 常见编程错误 2.23

如果 while、for 或 do/while 结构中的循环条件永远无法变成 false，则会造成无限循环。为了防止无限循环，一定要保证循环首部或循环体中某个地方的条件值改变，使循环条件最终能变为 false。

图 2.24 所示的程序用 do/while 重复结构打印数字 1 到 10。注意控制变量 counter 在循环条件测试中是前置自增的。另外，只有一个语句的 do/while 结构也使用了花括号。

```
1 // Fig. 2.24: fig02_24.cpp
2 // Using the do/while repetition structure
3 #include <iostream.h>
4
5 int main()
6 {
7     int counter = 1;
8
9     do {
10         cout << counter << " ";
11     } while ( ++counter <= 10 );
12
13     cout << endl;
14
15     return 0;
16 }
```

输出结果：

1 2 3 4 5 6 7 8 9 10

图 2.24 使用 do/while 重复结构

do/while 重复结构如图 2.25。这个流程图显示循环条件要在至少进行一次操作之后才执行。注意，流程图（除了小圆框和流程之外）也只能包含矩形框和菱形框，这是我们强调的操作/判断编程模型。程序员的任务就是根据算法需要用堆栈和嵌套两种方法组合其他几种控制结构，然后填入算法所要的操作和判断，从而生成程序。

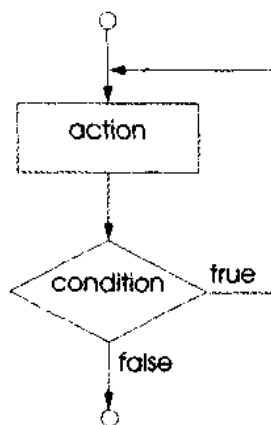


图 2.25 do/while 重复结构流程图

## 2.18 break 和 continue 语句

break 和 continue 语句改变控制流程。break 语句在 while、for、do/while 或 switch 结构中执行时，使得程序立即退出这些结构，从而执行该结构后面的第一条语句。break 语句常用于提前从循环退出或跳过 switch 结构的其余部分（如图 2.22）。图 2.26 演示了 for 重复结构中的 break 语句，if 结构发现 x 变为 5 时执行 break，从而终止 for 语句，程序继续执行 for 结构后面的 cout 语句。循环只执行四次。

注意这个程序中控制变量 x 在 for 结构首部之外定义。这是因为我们要在循环体中和循环执行完毕之后使用这个控制变量。

continue 语句在 while、for 或 do/while 结构中执行时跳过该结构体的其余语句，进入下一轮循环。在 while 和 do/while 结构中，循环条件测试在执行 continue 语句之后立即求值。在 for 结构中，执行递增表达式，然后进行循环条件测试。前面曾介绍过，while 结构可以在大多数情况下取代 for 结构。但如果 while 结构中的递增表达式在 continue 语句之后，则会出现例外。这时，在测试循环条件之前没有执行递增，并且 while 与 for 的执行方式是不同的。图 2.27 在 for 结构中用 continue 语句跳过该结构的输出语句，进入下一轮循环。

```

1 // Fig. 2.26: fig02_26.cpp
2 // Using the break statement in a for structure
3 #include <iostream.h>
4
5 int main()

```

```

6 {
7     // x declared here so it can be used after the loop
8     int x;
9
10    for ( x = 1; x <= 10; x++ ) {
11
12        if ( x == 5 )
13            break;    // break loop only if x is 5
14
15        cout << x << " ";
16    }
17
18    cout << "\nBroke out of loop at x of " << x << endl;
19    return 0;
20}

```

**输出结果:**

```

1 2 3 4
Broke out of loop at x of 5

```

图 2.26 for 重复结构中的 break 语句

```

1 // Fig. 2.27: fig02_07.cpp
2 // Using the continue statement in a for structure
3 #include <iostream.h>
4
5 int main()
6 {
7     for ( int x = 1; x <= 10; x++ ) {
8
9         if ( x == 5 )
10            continue; // skip remaining code in loop
11                        // only if x is 5
12
13        cout << x << " ";
14    }
15
16    cout << "\nUsed continue to skip printing the value 5"
17        << endl;
18    return 0;
19 }

```

**输出结果:**

```

1 2 3 4 5 6 7 8 9 10
Used continue to skip printing the value 5

```

图 2.27 在 for 结构用 continue 语句

### 编程技巧 2.30

有些程序员认为 break 和 continue 会破坏结构化编程。由于这些语句可以通过后面要介绍的结构化编程方法实现，因此这些程序员不用 break 和 continue。

### 性能提示 2.9

正确使用 break 和 continue 语句能比后面要介绍的通过结构化编程方法的实现速度更快。



## 软件工程视点 2.10

达到高质量软件工程与实现最佳性能软件之间有一定冲突。通常,要达到一个目标,就要牺牲另一个目标。

## 2.19 逻辑运算符

前面只介绍了 `counter <= 10`、`total > 1000` 和 `number != sentinelValue` 之类的简单条件 (simple condition)。我们用关系运算符 `>`、`<`、`>=`、`<=` 和相等运算符 `==`、`!=` 表示这些条件。每个判断只测试一个条件。要在每个判断中测试多个条件,可以在不同语句中或嵌套 `if (if/else)` 结构中进行这些测试。

C++ 提供的逻辑运算符 (logical operator) 可以用简单条件组合成复杂条件。逻辑运算符有逻辑与 (`&&`)、逻辑或 (`||`) 和逻辑非 (`!`)。下面将举例说明。

假设要保证两个条件均为 `true` 之后再选择某个执行路径,这时可以用 `&&` 逻辑运算符:

```
if ( gender == 1 && age >= 65 )
    ++seniorFemales;
```

这个 `if` 语句包含两个简单条件。条件 `gender==1` 确定这个人是男是女,条件 `age >=65` 确定这个人是否为老年人。`&&` 逻辑运算符左边的简单条件先求值,因为 `==` 的优先级高于 `&&`。如果需要,再对 `&&` 逻辑运算符右边的简单条件求值,因为 `>=` 的优先级高级于 `&&` (稍后将会介绍, `&&` 逻辑运算符右边的条件只在左边为 `true` 时才求值)。然后 `if` 语句考虑下列组合条件:

```
gender == 1 && age >= 65
```

如果两边的简单条件均为 `true`,则这个条件为 `true`。最后,如果这个条件为 `true`,则 `seniorFemales` 递增 1。如果两边的简单条件有一个为 `false`,则程序跳过该递增,处理 `if` 后面的语句。上述组合条件可以通过增加多余的括号而变得更加清楚:

```
( gender == 1 ) && ( age >= 65 )
```

## 常见编程错误 2.24

尽管 `3 < x < 7` 条件在数学上是正确的,但在 C++ 中无法正确求值,应改成 `( 3 < x && x < 7 )`。

图 2.28 的表格总结了 `&&` 逻辑运算符。表中显示了表达式 1 和表达式 2 的四种 `false` 和 `true` 值组合,这种表称为真值表 (truth table)。C++ 对所有包括逻辑运算符、相等运算符和关系运算符的所有表达式求值为 `false` 或 `true`。

表达式 1	表达式 2	表达式 1 && 表达式 2
false	false	false
false	true	false
true	false	false
true	true	true

图 2.28 逻辑与 (`&&`) 运算符的真值表

## 可移植性提示 2.5

为了与旧版 C++ 标准兼容, `bool` 值为 `true` 表示任何非 0 值, `bool` 值为 `false` 表示 0 值。

下面考虑逻辑或运算符 (`||`)。假设要保证两个条件至少有一个为 `true` 之后再选择某个执行路径,这时可以用逻辑或运算符,如下列程序段所示:

```
if (semesterAverage >= 90 || finalExam >= 90 )
    cout << "Student grade is A" << endl;
```

上述条件也包含两个简单条件。条件 `semesterAverage >= 90` 确定学生整个学期的表现是否为“A”，条件 `finalExam >= 90` 确定该生期末考试成绩是否优秀。然后 `if` 语句考虑下列组合条件：

```
semesterAverage >= 90 || finalExam >= 90
```

如果两个简单条件至少有一个为 `true`，则该生评为“A”。注意只有在两个简单条件均为 `false` 时才不会打印消息“Student grade is A”。图 2.29 是逻辑或 (`||`) 运算符的真值表。

`&&` 逻辑运算符的优先级高于 `||` 运算符。这两个运算符都是从左向右结合。包含 `&&` 和 `||` 运算符的表达式只在知道真假值时才求值。这样，求值下列表达式：

```
gender == 1 && age >= 65
```

将在 `gender` 不等于 1 时立即停止（整个表达式为 `false`），而 `gender` 等于 1 时则继续求值（整个表达式在 `age >= 65` 为 `true` 时为 `true`）。

表达式 1	表达式 2	表达式 1 && 表达式 2
false	false	false
false	true	true
true	false	true
true	true	true

图 2.29 逻辑或 (`||`) 运算符真值表

#### 常见编程错误 2.25

在使用 `&&` 逻辑运算符的表达式中，一个条件（称为相关条件）要在另一个条件为 `true` 时才有求值的必要。这时，相关条件应放在另一条件的后面，否则可能发生错误。

#### 性能提示 2.10

在使用 `&&` 逻辑运算符的表达式中，如果两个条件是相互独立的，则应把 `false` 可能性较大的条件放在左边；在使用 `||` 逻辑运算符的表达式中，应把 `true` 可能性较大的条件放在左边。这样可以减少程序执行时间。

C++ 提供了逻辑非 (`!`) 运算符，使程序员可以逆转条件的意义。与 `&&` 和 `||` 运算符不同的是，`&&` 和 `||` 运算符组合两个条件（是二元运算符），而逻辑非 (`!`) 运算符只有一个操作数（是一元运算符）。逻辑非 (`!`) 运算符放在条件之前，在原条件（不带逻辑非运算符的条件）为 `false` 时选择执行路径，例如：

```
if ( ! ( grade == sentinelValue) )
    cout << "The next grade is" << grade << endl;
```

条件 `grade == sentinelValue` 周围的括号是必需的，因为逻辑非 (`!`) 运算符的优先级高于 `==` 运算符。图 2.30 显示了逻辑非 (`!`) 运算符的真值表。

表达式	!表达式
false	true
true	false

图 2.30 逻辑非 (`!`) 运算符真值表

大多数情况下，程序员可以用相关的关系或相等运算符表达条件，避免使用逻辑非 (`!`) 运算符。例如，上述语句可以改写成如下形式：

```
if ( grade != sentinelValue)
    cout << "The next grade is" << grade << endl;
```

这种灵活性有助于程序员以更自然或更方便的方式表达条件。

图 2.31 显示了前面介绍的 C++ 运算符的优先级和结合律。运算符优先级从上到下逐渐降低。

运算符	结合律	类型
()	从左向右	括号
++    --    +    -    !    static_cast<type>()	从右向左	一元
*    /    %	从左向右	乘
+    -	从左向右	加
<<    >>	从左向右	插入/读取
<    <=    >    >=	从左向右	关系
==    !=	从左向右	相等
&&	从左向右	逻辑与
	从左向右	逻辑或
?:	从右向左	条件
=    +=    -=    *=    /=    %=	从右向左	赋值
,	从左向右	逗号

图 2.31 运算符优先级和结合律

## 2.20 混淆相等 (==) 与赋值 (=) 运算符

这是 C++ 程序员常见的错误，包括熟练的 C++ 程序员也会把相等 (==) 与赋值 (=) 运算符相混淆。这种错误的破坏性在于它们通常不会导致语法错误，而是能够顺利编译，程序运行完后，因为运行时的逻辑错误而得到错误结果。

C++ 有两个方面会导致这些问题。一个是任何产生数值的表达式都可以用于任何控制结构的判断部分。如果数值为 0，则当作 false，如果数值为非 0，则当作 true。第二是 C++ 赋值会产生一个值，即赋值运算符左边变量所取得的值。例如，假设把下列代码：

```
if ( payCode == 4 )
    cout << "You get a bonus!" << endl;
```

误写成如下形式：

```
if ( payCode = 4 )
    cout << "You get a bonus!" << endl;
```

第一个 if 语句对 paycode 等于 4 的人发奖金。而第二个 if 语句则求值 if 条件中的赋值表达式为常量 4。由于非 0 值解释为 true，因此这个 if 语句的条件总是 true，则人人都获得一份奖金，不管其 paycode 为多少。更糟的是，本来只要检查 paycode，却已经修改了 paycode。

### 常见编程错误 2.26

用 == 运算符进行赋值或用 = 运算符表示相等是个逻辑错误。

### 测试与调试提示 2.1

程序员通常将条件写成 `x == 7`，即变量名在左边，常量在右边。如果反过来，即变量名在右边，常量在左边，写成 `7 == x`，则编译器能防止程序员误把 == 运算符写成 =。编译器把这个错误当作语法错误，因为赋值语句左边只能放变量名，这样至少可以防止运行时逻辑错误的潜在破坏。

变量名可称为左值 (lvalue), 因为它可以放在赋值运算符左边。常量称为右值 (rvalue), 因为它只能放在赋值运算符右边。注意, 左值可以作为右值, 但右值不能用作左值。

还有一个问题也同样麻烦。假设程序员要用下列简单语句给一个变量赋值:

```
x = 1;
```

但却写成:

```
x == 1;
```

这也不是语法错误, 编译器只是求值条件表达式。如果  $x$  等于 1, 则条件为 true, 表达式返回 true 值。如果  $x$  不等于 1, 则条件为 false, 表达式返回 false 值。不管返回什么值都没有赋值运算符, 因此这个值丢失,  $x$  值保持不变, 有可能造成执行时的逻辑错误。但这个问题没有简单的解决办法。

#### 测试与调试提示 2.2

用文本编辑器搜索程序中的所有 =, 检查是否为正确的运算符。

## 2.21 结构化编程小结

建筑师设计大楼时, 要采用前人的智慧, 程序员设计程序时也要采用前人的智慧。我们的领域比建筑领域要年轻, 我们的集体智慧也比较少。前面曾介绍过, 结构化编程产生的程序比非结构化编程的程序更容易理解, 因此更容易测试、调试与修改, 并在数学意义上更加正确。

图 2.32 总结了 C++ 控制结构。图中的小圆表示每个结构的单入口点和单出口点。任意连接各个流程图符号可能造成非结构化编程。因此, 编程专业人员选择用流程图符号组成有限的控制结构, 通过用两种简单方法组合控制结构而建立结构化程序。

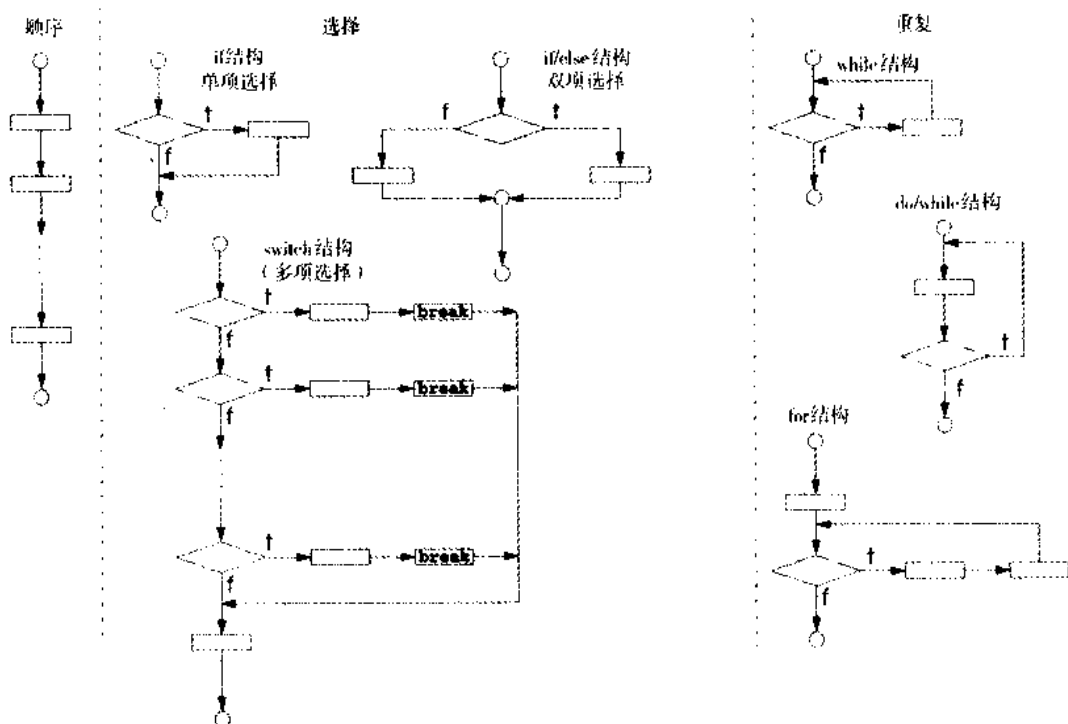


图 2.32 C++ 的单人/单出顺序、选择和重复结构

为了简单起见,我们只用单人/单出控制结构,每个控制结构只有一个入口和一个出口。顺序连接控制结构以形成简单的结构化程序,一个控制结构的出口连接下一个控制结构的入口,即控制结构只是在程序中一个接一个地放置,称为控制结构堆栈形式。形成结构化程序的规则还允许嵌套控制结构。

图 2.33 列出了正确形成结构化程序的规则。这个规则假设可以用流程图中的矩形框表示任何操作,包括输入/输出。

#### 形成结构化程序的规则

- 1) 从“最简单的流程图”开始(如图 2.34 所示)。
- 2) 任何矩形框(操作)可以换成两个顺序矩形框(操作)。
- 3) 任何矩形框(操作)可以换成任何控制结构(顺序、if、if/else、switch、while、do/while 或 for)。
- 4) 可按任何顺序多次重复规则 2 和规则 3。

图 2.33 形成结构化程序的规则

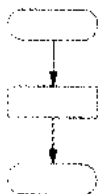


图 2.34 最简单的流程图

利用图 2.33 所示的规则总是可以得到整洁的结构化流程图。例如,对最简单的流程图重复采用规则 2 即可得到包含许多顺序放置矩形框的流程图(如图 2.35 所示)。注意规则 2 产生控制结构堆栈,因此称为堆栈规则(stacking rule)。

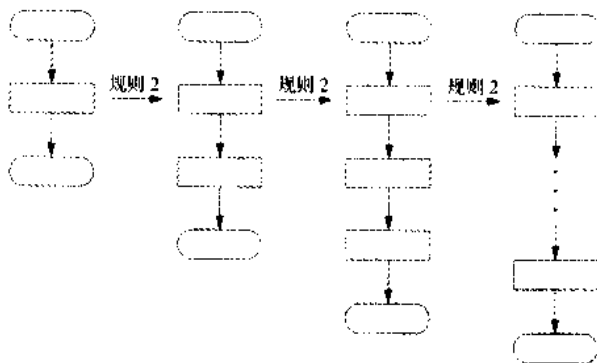


图 2.35 最简单的流程图重复采用规则 2

规则 3 称为嵌套规则(nesting rule)。对最简单的流程图重复采用规则 3 即可得到包含整齐嵌套控制结构的流程图。例如,图 2.36 中,首先将最简单的流程图中的矩形框换成双项选择结构(if/else)。然后再对双项选择结构中的两个矩形框采用规则 3,将每个矩形框变成一个双项选择结构。每个双项选择结构周围的虚线框表示最初的简单流程图中被替换的矩形框。

规则 4 产生更大、更复杂且层次更多的嵌套结构。文中出现的流程图采用图 2.33 的规则,构成各种可能的结构化流程图,从而设置各种可能的结构化程序。

结构化方法的妙处在于我们只用两种简单方法组合七种简单的单人/单出块。图 2.37 显示了采用规则 2 的堆栈构件块和采用规则 3 的嵌套构件块。图中还显示了结构化流程图中不能出现的重叠构件块(因为要消除 goto 语句)。

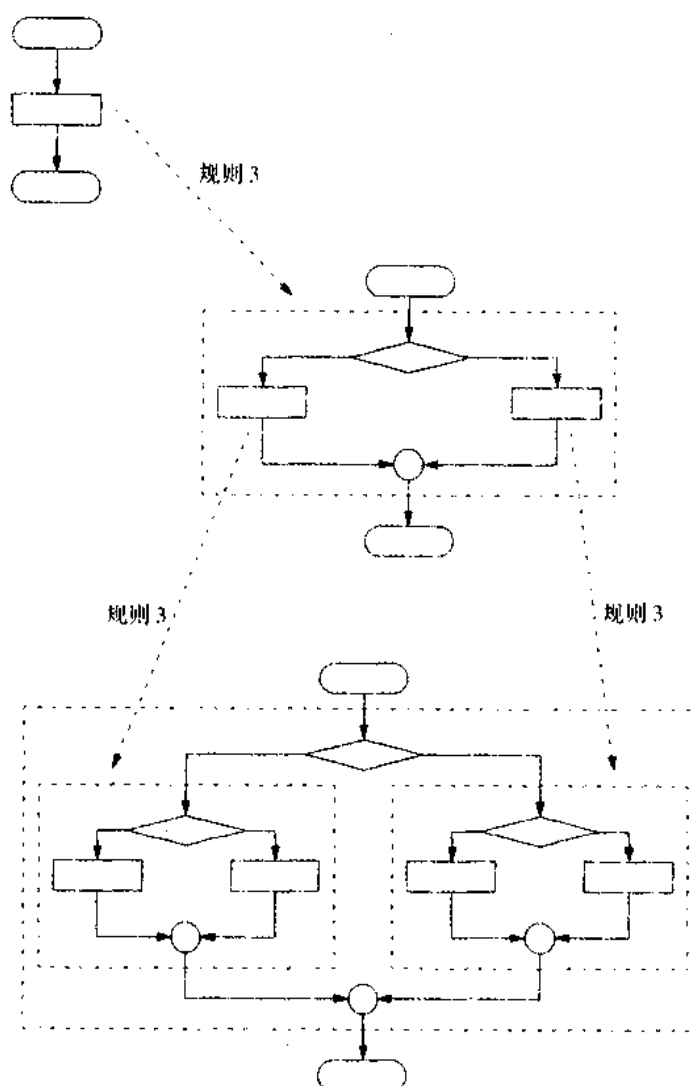


图 2.36 最简单的流程图重复采用规则 3

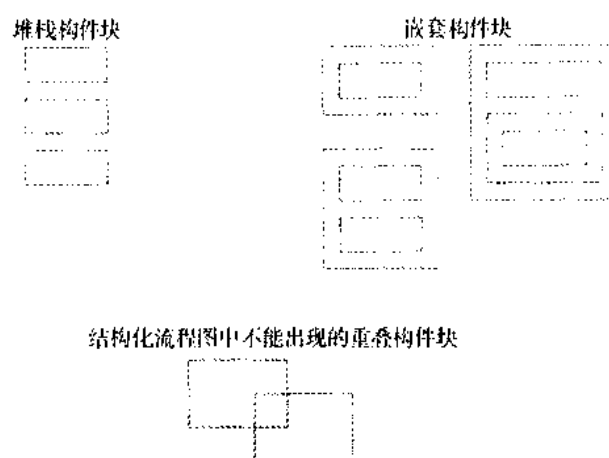


图 2.37 堆栈、嵌套和重叠构件块

如果遵照图 2.33 的规则, 则不会生成图 2.38 所示的非结构化流程图。如果不能判断某个流程图是否结构化, 可以逆向采用图 2.33 的规则, 将这个流程图简化为最简单的流程图。如果能够将这个流程图简化为最简单的流程图, 则原流程图是结构化流程图, 否则不是结构化流程图。



图 2.38 非结构化流程图

结构化编程提倡简单性。Bohm 和 Jacopini 已经证明, 只需要三种控制形式:

- 顺序 (sequence)
- 选择 (selection)
- 重复 (repetition)

顺序结构很简单, 选择可以用三种方法实现:

- if 结构 (单项选择)
- if/else 结构 (双项选择)
- switch 结构 (多项选择)

事实上很容易证明简单的 if 结构即可提供任何形式的选择, 任何能用 if/else 结构和 switch 结构完成的工作, 也可以组合简单 if 结构来实现 (但程序可能不够流畅)。

重复可以用三种方法实现:

- while 结构
- do/while 结构
- for 结构

很容易证明简单的 while 结构即可提供任何形式的重复, 任何能用 do/while 和 for 结构完成的工作, 也可以组合简单 while 结构来实现 (但程序可能不够流畅)。

根据以上结果, C++ 程序中所需的任何控制形式均可以用下列形式表示:

- 顺序
- if 结构 (选择)
- while 结构 (重复)

这些控制结构只要用两种方式组合, 即嵌套和堆栈。事实上, 结构化编程提倡简单性。

本章介绍了如何将包含操作和判断的控制结构组合成程序。第3章将介绍另一个程序结构单元——函数 (function), 我们将介绍如何用函数组成更大的程序 (函数也是由控制结构组成的) 以及函数如何提高软件复用性。第6章将介绍 C++ 的另一个程序结构单元——类 (class), 从类生成对象, 并进行面向对象编程。下面通过用面向对象设计攻克的难题来继续介绍对象。

## 2.22 有关对象的思考：确定问题中的对象

本节和“有关对象的思考”的下几节将介绍建立电梯模拟程序时的有趣问题和实际遇到的挑战。

第2章到第5章要完成面向对象设计（OOD）的各个步骤。从第6章开始，我们要用C++面向对象编程（OOP）技术实现电梯模拟程序。目前这些应用比较复杂，但不必在意，这章只考虑这个问题的一个小部分。

### 问题

公司要建立一幢两层的办公大楼并装上“最新”的电梯。公司要求开发一个面向对象的软件模拟程序，模拟电梯的操作，确定这个电梯能否满足需要。

这个电梯只限承载一人，为了省电，只在需要时才使用，电梯每天在一楼关门等待。

模拟程序包括一个时钟，每天从时间0开始，每秒滴答一次。模拟程序的调度器组件随机设置每一层第一个人到来的时间（第3章将介绍如何随机调度）。当时钟的时间等于第一个人到来的时间时，模拟程序对指定层生成一个新到的人并将人放在这一层。然后这个人按下该层的按钮，请求电梯开门。这个人的目的地楼层不能与他上电梯时所在的那层相同。

如果第1个人到达第1层，则他可以在按下按钮和等待电梯开门之后立即进入电梯。如果第1个人在第2层，则电梯要升到第2层去接这个人。电梯从1层移到另一层需要5秒钟。

电梯到达一层时，打开该层的电梯门上面的灯，并在电梯内发出铃声。该层的按钮和电梯中表示该层的按钮复位，电梯门打开，乘客（如果有人要乘电梯到该层）走出电梯，另一乘客（如果该层有人等待）进入电梯并按下目的地楼层的按钮，电梯门关上。如果电梯要开始移动，则要确定移动的方向（对只有两层的电梯，判断很简单）并移到下一层。为了简单起见，假设电梯到达一层时发生所有事件，而且直到电梯门关上所花的时间为0。电梯总是知道在哪层和要到哪层。

任何时间每层最多只能有一个人等待，如果新到的人（不在电梯中的人）要到达一层时该层已被占用，则一秒后才能安排新的到达者。假设每隔5到20秒人们随机到达每层，第3章将介绍如何用随机数产生器模拟到达每层的概率。

我们的目标是实现一个能够工作的软件模拟程序，并根据这些要求运行。该程序应模拟几分钟的电梯操作，确定电梯能否满足这座办公大楼的交通需求。

### 电梯实验室任务 1

在这些电梯实验室任务中，要进行面向对象设计的各个步骤。第一步要确定问题中的对象，最终要正式描述这些对象并在C++中实现。在这个电梯实验室任务中应该：

1. 确定这个电梯模拟问题中的对象。这个问题指定了许多对象模拟电梯以及与各个人、楼层、按钮等等之间的交互。找到问题中的名词（noun），这些名词通常就是实现电梯模拟问题中的对象。
2. 对每个找到的对象，用一段话描述关于该对象的所有事实。

### 说明

1. 这是个很好的小组练习，最好几个人一起讨论，小组成员之间可以互相提示，检讨和完善各人的设计和实现方法。
2. 小组应与班级中的其他小组一起竞赛，开发出最佳设计和实现方法。
3. 下一章介绍如何实现随机性，介绍随机数产生器。随机数产生器可以帮助读者模拟扔钱币和投骰子等，还可以模拟随机到达的乘电梯者。



4. 我们做出了许多简化的假设, 读者可以提供其他细节。
5. 由于现实世界是面向对象的, 因此完全可以在正式学习面向对象之前考虑这个项目。
6. 不要怕设计不完美。系统设计不是完善和完整的过程, 因此只要尽力而为即可。

### 问题

1. 如何确定电梯能否处理所需的交通量?
2. 为什么实现三层 (或更高) 的楼层时更加复杂?
3. 稍后会介绍, 建立一个电梯对象之后很容易建立更多的电梯对象。如果有多个电梯, 每个电梯在每一层载客和下客时, 都会遇到什么问题?
4. 为了简单起见, 我们指定电梯和每层的容量为一个乘客。如果增加这个容量, 会遇到什么问题?

### 小结

- 根据所要执行的操作和顺序解决问题的过程称为算法。
- 指定计算机程序执行语句的顺序称为程序控制。
- 伪代码是人为的非正式语言, 帮助程序员开发算法。
- 声明是种消息, 告诉编译器变量名和属性, 指示编译器在内存中为这个变量保留内存空间。
- 选择结构用于在多个操作之间进行选择。
- if 选择结构在条件为 true 时执行一个操作, 在条件为 false 时跳过这个操作。
- if/else 选择结构在条件为 true 或 false 时各执行一个操作。
- 要在 if 选择结构体中包括多条语句, 就要把这些语句放在花括号中。复合语句可以放在程序中出现单个语句的任何地方。
- 空语句就是在正常语句出现的地方放一个分号 (;), 表示不采取任何操作。
- 重复结构 (repetition structure) 使程序员可以指定一定条件下可以重复的操作。
- while 重复结构的形式为:

```
while ( conditon)
    statement
```

- 包含小数的值称为浮点数, 用数据类型 float 表示。
- C++ 包括一元强制类型转换运算符 static\_cast< float >(), 生成用于计算的临时浮点数值。
- C++ 提供算术赋值运算符 +=、-=、\*=、/= 和 %=, 能够缩写某些常用类型的表达式。
- C++ 提供自增 (++) 和自减 (--) 运算符, 可以将变量加 1 或减 1。如果运算符放在变量前面, 则变量先加 1 (减 1), 然后在表达式中使用。如果运算符放在变量后面, 则变量先在表达式中使用, 然后加 1 (减 1)。
- 循环是一组计算机指令重复执行, 直到符合某个终止条件。其中有两种重复是计数器控制重复和标记控制重复。
- 循环计数器计算一组指令的重复次数。每次执行这组指令时, 其通常加 1 或减 1。
- 标记值通常用于控制事先不知道重复次数的循环, 循环中包括每次执行循环时读取数据的语句。标记值在所有有效数据之后输入, 应不同于有效数据项目。
- for 重复结构处理计数器控制循环的所有细节。for 结构的一般格式如下:

```
for (expression1; expression2; expression3)
    statement
```

其中 expression1 初始化循环控制变量, expression2 是循环条件, expression3 递增控制变量。

- do/while 重复结构执行循环体之后再测试循环条件, 因此, do/while 结构至少执行循环体一次。do/while 结构的形式如下:

```
do
    statement
while (condition);
```

- break 语句在 while、for 和 do/while 结构中执行时, 将使程序立即退出这些结构。
- continue 语句在 while、for 或 do/while 结构中执行时跳过该结构体的其余语句, 进入下一轮循环。
- switch 语句处理一系列判断, 测试特定变量或表达式的值并相应地采取不同操作。大多数程序中, 每个 case 后面的语句之后要包括一个 break 语句。几个 case 可以执行相同语句, 只要在语句前面列出这些 case 标号。switch 结构只能测试常量整型表达式。多个 case 语句不必放在花括号中。
- 在 UNIX 系统和其他许多系统中, 表示文件结束符的输入如下:

<ctrl-d>

而在 VMS 和 DOS 系统中, 表示文件结束符的输入如下:

<ctrl-z>

- 逻辑运算符可以用简单条件组合成复杂条件。逻辑运算符有逻辑与 (&&)、逻辑或 (||) 和逻辑非 (!)。
- true 值可以表示任何非 0 值, false 值也可以表示 0 值。

## 术语

&& operator    && 运算符

|| operator    || 运算符

! operator    !运算符

-- operator    -- 运算符

++ operator    ++ 运算符

?: operator    ?: 运算符

action/decision model    操作 / 判断模型

algorithm    算法

arithmetic assignment operators: =+, -=, \*=, /=

和 %=    算术赋值运算符

ASCII character set    ASCII 字符集

block    块 (程序块)

body of a loop    循环体

bool

break

case label    case 标号

cast operator    强制类型转换运算符

char

cin.get() function    cin.get()函数

compound statement    复合语句

conditional operator (?)    条件运算符

continue

control structure    控制结构

counter-controlled repetition    计数器控制重复

decrement operator (--)    自减运算符

default case in switch    switch 中的默认 case

definite repetition    确定重复

definition    定义

delay loop    延迟循环

do/while repetition structure    do/while 重复结构

double

double-selection structure    双项选择结构

empty statement (;)    空语句

EOF

false	postincrement operator 后置自增运算符
fatal error 致命错误	pow function pow 函数
filed width 域宽	predecrement operator 前置自减运算符
fixed-point format 定点格式	preincrement operator 前置自增运算符
float	pseudocode 伪代码
for repetition structure for 重复结构	repetition 重复
garbage value 垃圾值	repetition structures 重复结构
if selection structure if 选择结构	rvalue(“right value”) 右值
if/else selection structure if/else 选择结构	selection 选择
increment operator(++) 自增运算符	sentinel value 标记值
indefinite repetition 不确定重复	sequential execution 顺序执行
infinite loop 无限循环	setiosflags stream manipulator setiosflags 流操纵算子
initialization 初始化	setprecision stream manipulator setprecision 流操纵算子
integer division 整除	setw stream manipulator setw 流操纵算子
ios::fixed	short
ios::left	single-entry/single-exit control structures 单人/单出控制结构
ios::showpoint	single-selection structure 单项选择结构
keyword 关键字	stacked control structures 堆栈控制结构
logic error 逻辑错误	static_cast<type>()
logical AND(&&) 逻辑与	structured programming 结构化编程
logical negation(!) 逻辑非	switch selection structure switch 选择结构
logical operators 逻辑运算符	syntax error 语法错误
logical OR(  ) 逻辑或	ternary operator 三元运算符
long	top-down, stepwise refinement 自上而下逐步完善
loop counter 循环计数器	transfer of control 控制转移
looping 循环	true
loop-continuation condition 循环条件	unary operator 一元运算符
lvalue(“left value”) 左值	undefined value 未定义数值
multiple-selection structure 多项选择结构	while repetition structure while 重复结构
nested control structures 嵌套控制结构	whitespace characters 空白字符
nonfatal error 非致命错误	
off-by-one error 差1错误	
parameterized stream manipulator 参数化流操纵算子	
postdecrement operator 后置自减运算符	

## 自测练习

练习 2.1 到 2.10 对应于 2.1 节到 2.12 节。练习 2.11 到 2.13 对应于 2.13 节到 2.21 节。

2.1 填空:

a) 所有程序均可用三种控制结构编写: \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。

- b) \_\_\_\_\_ 选择结构用于在条件为 true 时执行一个操作, 条件为 false 时执行另一个操作。
- c) 将一组指令重复特定次数称为 \_\_\_\_\_ 重复。
- d) 事先不知道一组指令重复次数时, 可以用 \_\_\_\_\_ 值终止重复。
- 2.2 编写四种不同的 C++ 语句, 对整数变量 x 加 1。
- 2.3 编写完成下列任务的 C++ 语句:
- a) 将 x 和 y 的和赋给 z, 并在计算之后将 x 的值加 1。
  - b) 测试变量 count 的值是否大于 10。如果是, 则打印 "Count is greater than 10"。
  - c) 将变量 x 减 1, 然后将 total 变量的值减去 x。
  - d) 计算 q 除以 divisor 的余数, 并将结果赋给 q。用两种方法编写这个语句。
- 2.4 编写完成下列任务的 C++ 语句:
- a) 将变量 sum 和 x 声明为 int 类型。
  - b) 将变量 x 初始化为 1。
  - c) 将变量 sum 初始化为 0。
  - d) 将 x 加进 sum 中, 并将结果赋给 sum。
  - e) 打印 "The sum is:" 加上 sum 的值。
- 2.5 将练习 2.4 的语句合并为一个程序计算和打印从 1 到 10 的整数的和。利用 while 结构在计算和递增语句之间循环, 循环在 x 为 11 时终止。
- 2.6 确定计算完成后每个变量的值。假设每个语句开始执行时, 所有变量为整数值 5。
- a) `product *= x++;`
  - b) `quotient /= ++x;`
- 2.7 编写一条 C++ 语句:
- a) 用 cin 和 >> 输入整型变量 x。
  - b) 用 cin 和 >> 输入整型变量 y。
  - c) 将整型变量 i 初始化为 1。
  - d) 将整型变量 power 初始化为 1。
  - e) 将变量 power 乘以 x 并将结果赋给 power。
  - f) 将变量 i 加 1。
  - g) 测试 i 是否小于或等于 y。
  - h) 用 cout 和 << 输出整型变量 power。
- 2.8 编写一个 C++ 程序, 用练习 2.7 的语句计算 x 的 y 次方。程序采用 while 重复控制结构。
- 2.9 判断下列语句的对错, 并纠正其中的错误。
- a) 

```
while (c <= 5) {  
    product *= c;  
    ++c;  
}
```
  - b) `cin << value;`
  - c) 

```
if (gender == 1)  
    cout << "Woman" << endl;  
else;  
    cout << "Man" << endl;
```
- 2.10 下列 while 结构错在哪里:
- ```
while ( z >= 0)  
    sum += z;
```

2.11 判断下列各题是否正确。如果不正确,请说明原因。

- a) switch 选择结构中必须有 default。
- b) switch 选择结构的 default 中必须有 break 语句才能正确退出这个结构。
- c) 如果表达式  $x > y$  为 true 或  $a < b$  为 true, 则表达式  $(x > y \ \&\& \ a < b)$  为 true。
- d) 如果至少有一个操作数为 true, 则包含  $\parallel$  运算符的表达式为 true。

2.12 编写一条或一组 C++ 语句, 完成下列任务:

- a) 用 for 结构求 1 到 99 的奇数和。假设已经声明整型变量 sum 和 count。
- b) 在域宽为 15 个字符、精度分别为 1、2、3 的输出域中打印数值 333.546372。在同行打印每个值, 在输出域中左对齐每个值。
- c) 用 pow 函数计算 2.5 的 3 次方。在域宽为 10 个字符、精度分别为 2 的输出域中打印计算结果。
- d) 用 while 循环和计数器变量 x 打印 1 到 10 的整数。假设变量 x 已经声明, 但还没有初始化。每行只打印 5 个整数。提示: 用算式  $x \% 5$ 。数值为 0 时, 打印换行符, 否则打印制表符。
- e) 用 for 结构重做练习 2.12d)。

2.13 寻找下列代码中的错误并说明如何纠正。

- a) 

```
x = 1
while (x <= 10);
    x++;
}
```
- b) 

```
for (y = .1; y != 1.0; y += .1)
    cout << y << endl;
```
- c) 

```
switch (n) {
    case 1:
        cout << "The number is 1" << endl;
    case 2:
        cout << "The number is 2" << endl;
        break;
    default:
        cout << "The number is not 1 or 2" << endl;
        break;
}
```
- d) 下列代码打印 1 到 10 的值。  

```
n = 1;
while (n < 10)
    cout << n++ << endl;
```

## 自测练习答案

- 2.1 a) 顺序、选择和重复。b) if/else。c) 计数器控制或确定。d) 标记 (记号、标志或哑元)。
- 2.2 

```
x = x + 1;
x += 1;
++x;
x++;
```
- 2.3 a) 

```
z = x++ + y;
```

  
b) 

```
if ( count > 10 )
```

- ```
        cout << "Count is greater than 10" << endl;
```
- c) total -= --x;
- d) q %= divisor;  
 q = q % divisor;
- 2.4 a) int sum, x;
- b) x = 1;
- c) sum = 0;
- d) sum += x; 或 sum = sum + x;
- e) cout << "The sum is:" << sum << endl;
- 2.5 如下所示:
- ```
1 // Calculate the sum of the integers from 1 to 10
2 #include <iostream.h>
3
4 int main()
5 {
6     int sum,x;
7     x = 1;
8     sum = 0;
9     while (x <= 10) {
10         sum += x;
11         ++x;
12     }
13     cout << "The sum is:" << sum << endl;
14     return 0;
15 }
```
- 2.6 a) product = 25, x = 6;
- b) quotient = 0, x = 6;
- 2.7 a) cin >> x;
- b) cin >> y;
- c) i = 1;
- d) power = 1;
- e) power \*= x; 或 power = power \* x;
- f) i++;
- g) if (i <= y)
- h) cout << power << endl;
- 2.8 如下所示:
- ```
1 // raise x to the y power
2 #include <iostream.h>
3 int main()
4 {
5     int x, y, i, power;
6
7     i = 1;
8     power = 1
9     cin >> x;
10    cin >> y;
11
12    while (i <= y) {
```

```

13     power *= x;
14     ++i;
15 }
16
17 cout << powe << endl;
18 return 0;
19 }

```

2.9 a) 不正确: while 循环缺右花括号。

纠正: 在 ++C; 语句后面加上右花括号。

b) 不正确: 用流插入而不是流读取运算符。

纠正: 将 << 变为 >>。

c) 不正确: else 后面的分号造成逻辑错误, 第二个输出语句总是会执行。

纠正: 将 else 后面的分号删除。

2.10 while 结构中变量 z 的值永远不变。因此如果循环测试条件 ( $z \geq 0$ ) 为 true, 则会生成无限循环。要防止生成无限循环, 必须递减 z, 最终达到小于 0。

2.11 a) 不正确。default 是可选的, 如果不需要默认操作, 则不需要 default。

b) 不正确。break 语句用于退出 switch 结构。如果 default 为最后一个 case, 则不需要 break 语句。

c) 不正确。使用 && 运算符的两个关系表达式都为 true 时, 整个表达式才能为 true。

d) 正确。

2.12 a) sum = 0

```

for ( count = 1; count <= 99; count += 2 )
    sum += count;

```

```

b) cout << setiosflags(ios::fixed | ios::showpoint | ios::left)
    << setprecision(1) << setw (15) << 333.546372
    << setprecision(2) << setw (15) << 333.546372
    << setprecision(3) << setw (15) << 333.546372
    << endl;

```

输出如下:

```

333.5          333.55          333.546

```

```

c) cout << setiosflags(ios::fixed | ios::showpoint)
    << setprecision(2) << setw (10) << pow(2.5,3)
    << endl;

```

输出如下:

```

15.63

```

d) x = 1;

```

while (x <= 20) {
    cout << x;
    if (x % 5 == 0)
        cout << endl;
    else
        cout << '\t';
    x++;
}

```

```

e) for (x = 1; x <= 20; x++) {
    cout << x;
    if (x % 5 == 0)

```

```

        cout << endl;
    else
        cout << '\t';
}
或
for (x = 1; x <= 20; x++)
    if (x % 5 == 0)
        cout << x << endl;
    else
        cout << x << '\t';

```

2.13 a) 不正确: while 首部后面的分号会导致无限循环。

纠正: 将分号换成 “{” 或删除 “;” 和 “}”。

b) 不正确: 用浮点数控制 for 重复结构。

纠正: 用整数, 并进行适当的计算以取得所要的值。

```

for ( y = 1; y != 10; y++ )
    cout << ( static_cast< float > (y) / 10 ) << endl;

```

c) 不正确: 第一个 case 的语句中缺少 break 语句。

纠正: 第一个 case 语句中补上 break 语句。注意, 如果程序员要让 case 1 语句每次执行时都执行 case 2 的语句, 则这不是错误。

d) 不正确: while 重复条件中使用不正确的关系运算符。

纠正: 用 “<=” 而不用 “<”, 或将 10 变为 11。

## 练习

练习 2.14 到 2.38 对应 2.1 节到 2.12 节。练习 2.39 到 2.63 对应 2.13 节到 2.21 节。

2.14 判断下列语句的对与错 (注意, 每个代码中可能有多处错误):

```

a) if ( age >= 65 );
    cout << "Age is greater than or equal to 65" << endl;
    else
        cout << "Age is less than 65 << endl";
b) if ( age >= 65 );
    cout << "Age is greater than or equal to 65" << endl;
    else
        cout << "Age is less than 65 << endl";
c) int x = 1, total;
    while ( x <= 10 ) {
        total += x;
        ++x;
    }
d) while (x <= 100 )
    total += x;
    ++x;
e) while (y > 0) {
    cout << y << endl;
    ++y
}

```

2.15 下列程序打印什么结果?

```

#include <iostream.h>
int main()

```



```
{
    int y, x = 1, total = 0;
    while (x <= 10) {
        y = x * x;
        cout << y << endl;
        total += y;
        ++x;
    }
    cout << "Total is" << total << endl;
    return 0;
}
```

对练习 2.16 到 2.19 完成下列步骤:

- a) 阅读问题。
- b) 用伪代码和自上而下逐步完善的方法构造算法。
- c) 编写 C++ 程序。
- d) 测试、调试和执行 C++ 程序。

2.16 驾驶员比较注意汽车的行驶里程。经常计算公里数和每箱油使用的加仑数。开发一个 C++ 程序,输入公里数和每箱油使用的加仑数,程序计算和显示每箱油每加仑跑出的公里数。处理所有输入信息后,程序应计算和打印所有油箱每加仑跑出的公里数组合。

```
Enter the gallons used (-1 to end): 12.8
Enter the miles driven:287
The miles / gallon for this tank was 22.421875

Enter the gallons used (-1 to end): 10.3
Enter the miles driven:200
The miles / gallon for this tank was 19.417475

Enter the gallons used (-1 to end): 5
Enter the miles driven:120
The miles / gallon for this tank was 24.000000

Enter the gallons used (-1 to end): -1

The overall average miles/gallon was 21.601423
```

2.17 开发一个 C++ 程序,确定商场客户支付的款额是否超过付款账号的信用额度。对每个客户,提供下列事实:

- a) 账号(整数)
- b) 月初结余
- c) 该客户当月购买的总物品数
- d) 该客户当月总付款金额
- e) 允许的信用额度

程序应输入每个信息,计算新结余(月初结余+付款-透支数),确定新结余是否超过付款账号的信用额度。如果客户的款额超过信用额度,则程序显示该客户的账号、信用额度、新结余和消息“Credit limit exceeded”。

```
Enter account number (-1 to end): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
```

```
Account: 100
Credit limit: 5500.00
Balance: 5894.78
Credit Limit Exceeded.

Enter account number (-1 to end): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00

Enter account number (-1 to end): 300
Enter beginning balance: 500.00
Enter total charges: 274.73
Enter total credits: 100.00
Enter credit limit: 800.00

Enter account number (-1 to end): -1
```

- 2.18 一家大型化工公司根据佣金向销售人员发工资。销售人员每周可取得 200 美元加上本周销售额的 9%。例如，如果本周销售额为 5000 美元，则本周收入为 200 美元加上 5000 美元的 9%，总共 650 美元。开发一个 C++ 程序，输入每个员工本周销售额，并计算和显示其总收入。一次处理一个销售人员的数据。

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 6000.00
Salary is: $740.00

Enter sales in dollars (-1 to end): 7000.00
Salary is: $830.00

Enter sales in dollars (-1 to end): -1
```

- 2.19 开发一个 C++ 程序，确定几个员工的总工资。公司对每个员工的前 40 个小时发计时工资，此后发加班工资（是原工资的 1.5 倍）。你可以得到一系列的公司员工名单、每个员工上周工作小时数和每个员工的小时工资。程序要输入每个员工的这些信息，确定和显示员工的总工资。

```
Enter hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter hours worked (-1 to end): -1
```

- 2.20 寻找最大数（即求出一组成员中的最大值）的过程经常在计算机应用程序中使用。例如，确定销售竞赛优胜者的程序要输入每个销售员销售的单位数，销售的单位数最多的员工为销售竞赛优胜者。编写一个伪代码程序，然后变成 C++ 程序，输入 10 个数，确定和打印其中的最大数。提示：程序应使用如下三个变量。

counter: 从1到10的计数器 (即跟踪输入了多少个数, 确定何时处理了这10个数)。

number: 当前输入程序的数。

largest: 到目前为止的最大数。

2.21 编写一个C++程序, 利用循环和制表转义序列“\t”打印下列数据表:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

2.22 用练习2.20的方法寻找10个数中的两个最大数。注意: 每个数只允许输入一次。

2.23 修改图2.11以验证其输入。对任何输入, 如果输入的值不是1和2就一直保持循环, 直到用户输入正确的值。

2.24 下列程序打印什么结果?

```
#include <iostream.h>
int main()
{
    int count = 1;
    while ( count <= 10) {
        cout << ( count % 2 ? "*****": "+++++++" )
            << endl;
        ++count;
    }
    return 0;
}
```

2.25 下列程序打印什么结果?

```
#include <iostream.h>
int main()
{
    int row = 10, column;
    while (row >= 1) {
        column = 1;
        while ( column <= 10 ) {
            cout << (row % 2 ? "<" : ">" );
            ++column;
        }
        --row;
        cout << endl;
    }
    return 0;
}
```

2.26 (悬挂else问题) 确定x为9和y为11与x为11和y为9时下列各题的输出。注意, 编译器忽略C++程序中的缩排。另外, C++编译器总是将else与前面的if语句相关联, 除非用花括号({})另外指定。由于程序员初看时不知道哪个if与哪个else相配, 因此称为悬挂else问题。我们消除下列程序中的缩排, 使问题显得更复杂些。提示: 采用书中介绍的缩排方法。

a) if ( x < 10 )

```

    if ( y > 10 )
    cout << "*****" << endl;
    else
    cout << "#####" << endl;
    cout << "$$$$$" << endl;
b) if ( x < 10 ){
    if ( y > 10 )
    cout << "*****" << endl;
    }
    else {
    cout << "#####" << endl;
    cout << "$$$$$" << endl;
    }
}

```

2.27 (另一个悬挂 else 问题) 修改下列代码, 产生图中所示的输出。用正确的缩排方法, 除了插入花括号之外, 不能进行其他任何改变。编译器忽略 C++ 程序中的缩排。我们消除下列程序中的缩排, 使问题显得更复杂些。注意: 可以不进行其他任何改变。

```

    if ( y == 8 )
    if ( x == 5 )
    cout << "@@@@@" << endl;
    else
    cout << "#####" << endl;
    cout << "$$$$$" << endl;
    cout << "&&&&&" << endl;

```

a) 假设 x=5, y=8, 产生下列输出:

```

@@@@@
$$$$$
&&&&&

```

b) 假设 x=5, y=8, 产生下列输出:

```

@@@@@

```

c) 假设 x=5, y=8, 产生下列输出:

```

@@@@@
&&&&&

```

d) 假设 x=5, y=7, 产生下列输出。说明: else 后面的三个输出语句放在一个复合语句中。

```

#####
$$$$$
&&&&&

```

2.28 编写一个程序, 读取正方形长度, 然后打印一个由星号围成的空心正方形, 长度为指定长度。程序适用于边长为 1 到 20 的正方形。例如, 如果程序读取正方形长度 5, 则打印:

```

*****
*   *
*   *
*   *
*   *
*****

```

2.29 回文是指数字或文本正向和逆向读时都一样的文本。例如, 下列五位整数都是回文: 12321、55555、45554 和 11611。编写一个程序, 读取五位整数并确定其是否回文。提示: 用除法和求模运算符将数字分成各个位。

2.30 输入只包含 0 和 1 的整数 (如二进制整数), 并打印其十进制对应的值。提示: 用除法和求模运算符一次一位从右向左取得这个二进制整数的位。和十进制数值系统中一样, 最

右边的位值为1, 向左一位为10, 然后为100, 然后1000等等, 二进制系统中最右边的位值为1, 向左一位为2, 然后为4, 然后为8等等。这样, 十进制数234可以解释为 $4 \times 1 + 3 \times 10 + 2 \times 100$ 。二进制1101的十进制值为 $1 \times 1 + 0 \times 2 + 1 \times 4 + 1 \times 8$ 或 $1 + 0 + 4 + 8$ 即13)。

2.31 编写一个程序, 显示下列棋盘图案。

```

* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
 * * * * *

```

程序只能用三个输出语句, 其形式如下:

```

cout << " * ";
cout << ' ';
cout << endl;

```

2.32 编写一个程序, 打印2的倍数2、4、6、8、16、32、64等等。循环不能终止(即生成无限循环)。运行这个程序时会发生什么情况?

2.33 编写一个程序, 读取圆的半径(作为float类型的值)并计算和打印直径、周长和面积。 $\pi$ 取值3.14159。

2.34 下列语句有什么错误? 提供正确语句完成程序员的意图。

```
cout << ++ ( x + y );
```

2.35 编写一个程序读取三个float类型的非0值, 并确定和打印其能否成为三角形的三个边。

2.36 编写一个程序读取三个非0整数值, 并确定和打印其能否成为三角形的三个边。

2.37 公司要在电话线上传输数据, 但担心电话被窃听。所有数据用四位整数传输, 他们要求编写一个程序, 将数据进行加密, 使数据传输更安全。程序读取四位整数并加密如下: 将每个位换成该位与7的和并用10求模。然后交换第一位与第三位, 交换第二位与第四位, 并打印加密后的整数。再编写一个程序读取加密的四位数, 并解密成原先的四位数。

2.38 非负整数的阶乘写成 $n!$ (读作“n的阶乘”), 定义如下:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$  (n大于或等于1时)

和

$n! = 1$  (n=0时)

例如,  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$  等于120。

a) 编写一个程序, 读取这个非负整数值, 计算和打印其阶乘。

b) 编写一个程序, 估算数学常量e的值, 公式如下:

$$\frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

c) 编写一个程序, 计算 $e^x$ 的值, 公式如下:

$$\frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

2.39 寻找下列各题的错误(注意, 可能有多处错误):

```

a) For ( x = 100, x >= 1 x++
      cout << x << endl;

```

b) 下列代码打印整数value为奇数或偶数:

```

switch ( value % 2 ) {
    case 0:
        cout << "Even integer" << endl;
    case 1:
        cout << "odd integer" << endl;
}

```

c) 下列代码输出 19 到 1 的奇数:

```

for ( x = 19; x >= 1; x -= 2 )
    cout << x << endl;

```

d) 下列代码输出 2 到 100 的偶数:

```

counter = 2;
do {
    cout << counter << endl;
    counter += 2;
} While ( counter < 100 );

```

2.40 编写一个程序求一系列整数的和。假设第一个读取的整数指定后面要输入的数值个数，程序在每个输入语句中只读取一个值。典型的输入序列如下:

```
5 100 200 300 400 500
```

其中 5 表示指定后面要输入 5 个值。

2.41 编写一个程序计算和打印几个整数的平均值。假设最后一个读取值是标记 9999。典型的输入顺序如下:

```
10 8 11 7 9 9999
```

表示计算标记 9999 之前所有数的平均值。

2.42 下列程序有什么作用?

```

#include <iostream.h>
int main()
{
    int x,y;
    cout << "Enter two integers in the range 1-20:";
    cin >> x >> y;
    for ( int i = 1; i <= y; i++ ) {
        for ( int j = 1; j <= x; j++ )
            cout << '0';
        cout << endl;
    }
    return 0;
}

```

2.43 编写一个程序，寻找几个整数的最小值。假设第一个值指定后面要输入的数值个数。

2.44 编写一个程序，计算和打印 1 到 15 的奇数积。

2.45 阶乘函数常用于概率问题。正整数  $n$  的阶乘写成  $n!$  (读作“ $n$  的阶乘”) 等于从 1 到  $n$  的正整数的积。编写一个程序，求 1 到 5 的阶乘。将结果打印成表格形式。要计算 20 的阶乘会有什么困难?

- 2.46 修改第 2.15 节的复利程序，将利率变为 5%、6%、7%、8%、9% 和 10%。用 for 循环改变利率。
- 2.47 编写一个程序打印下列图案。使用 for 循环一个图案一个图案地打印。所有星号 (\*) 应在一条形式为 `cout << '*'` 的语句中打印 (使星号靠在一起)。提示：最后两个图案要求每一行以适当空格数开始。附加部分：将这 4 个问题的代码组合在一个程序中，利用嵌套 for 循环使 4 个图案并排打印。

[illegible]

- 2.48 计算机的一个有趣应用是绘制图形和条形图（也称直方图）。编写一个程序，读取5个数（1到30之间）。对读取的每个数，程序应打印相应这个数目的星号。例如，如果程序读取数字7，则打印\*\*\*\*\*。
- 2.49 邮购公司销售五种不同的零售产品，零售价分别为product 1—\$2.98、product 2—\$4.50、product 3—\$9.98、product 4—\$4.49和product 5—\$6.87。编写一个程序，读取如下一系列数值对：
- a) 产品号  
b) 一天销售量
- 程序要用switch语句确定每种产品的零售价并计算和显示上周所销售产品的总零售额。
- 2.50 修改图2.22的程序，使它计算班级平均成绩点，A为4点，B为3点等等。
- 2.51 修改图2.21的程序，只用整数计算复利。提示：把所有货币值当作整数，然后用除法和求模运算将结果分为美元和美分，中间用圆点分开。
- 2.52 假设i=1、j=2、k=3和m=2。下列语句输出什么结果？括号是否需要？
- a) `cout << ( i == 1 ) << endl;`  
b) `cout << ( j == 3 ) << endl;`  
c) `cout << ( i >= 1 && j < 4 ) << endl;`  
d) `cout << ( m >= 99 && k < m ) << endl;`  
e) `cout << ( j >= i || k < m ) << endl;`  
f) `cout << ( k + m < j || 3 - j >= k ) << endl;`  
g) `cout << ( !m ) << endl;`  
h) `cout << ( !( j - m ) ) << endl;`  
i) `cout << ( !( k > m ) ) << endl;`
- 2.53 编写一个程序。打印整数1到156的二进制、八进制、十六进制和十进制对照表。如果不熟悉这些进制，见附录C。
- 2.54 用下列数学表达式计算 $\pi$ 的值：

打印用1项、2项、3项等时  $\pi$  的近似值。要用多少项才能得到近似值3.14、3.141、3.1415、3.14159?

- 2.55 (直角三角形) 直角三角形的三条边都可以取整数, 如果两边平方的和等于第三边(弦)的平方, 则这个三角形是直角三角形。编写程序找出500以内直角三角形三边side1、side2和hypotenuse值。用三个嵌套for循环求出所有值, 这是个强制计算的例子。在高级的计算机学科中还有许多有趣的问题, 这些问题没有别的解决办法, 只能通过强制计算。
- 2.56 公司员工分为经理(每周发固定工资)、计时工(40小时之内固定小时工资, 40小时以外的工资按原工资1.5倍)、佣金工(250美元的基本工资加每月销售额的5.7%)和计件工(每生产一件产品发固定工资, 公司的计件工只生产一种产品)。编写一个程序, 计算每个员工每周的工资。事先不知道员工人数, 每种员工有自己的工资代码, 经理的工资代码为1, 计时工的工资代码为2, 佣金工的工资代码为3, 计件工的工资代码为4。用switch根据工资代码计算每个员工的工资。在switch中, 提示用户输入根据工资代码计算每个员工的工资时所需的信息。

- 2.57 (摩根定律) 本章介绍了逻辑运算符&&、||和!。摩根定律可以使逻辑表达式的表达更方便。摩根定律指出, 表达式!(condition 1 && condition 2)逻辑上等价于表达式(!condition 1 || condition 2), 同样表达式!(condition 1 || condition 2)逻辑上等价于表达式(!condition 1 && !condition 2)。利用摩根定律编写下列表达式的等价表达式, 然后编写一个程序, 显示原表达式和新表达式的等价性。

- a) `!( x < 5 ) && !( y >= 7 )`
- b) `!( a == b ) || !( g != 5 )`
- c) `!( ( x <= 8 ) && ( y > 4 ) )`
- d) `!( ( i > 4 ) || ( j <= 6 ) )`

- 2.58 编写一个程序, 打印下列菱形形状。可以用输出语句打印单个星号(\*)或单个空格。充分利用重复结构(用嵌套for结构), 减少输出语句个数。

```

      *
     ***
    *****
   *********
  *********
 *****
  *****
   *****
    *****
     *****
      *

```

- 2.59 修改练习2.58所写的程序, 读取1到19的奇数, 指定菱形形状的行数。然后程序显示相应的菱形形状。
- 2.60 有人认为break语句和continue语句是非结构化的。实际上, break语句和continue语句总是可以转换成结构化语句, 但这样做容易出问题。描述如何从程序循环中删除break语句并转换成结构化语句。提示: break语句从循环体中离开循环。另一种离开的方法是让循环条件测试失败。可以在循环条件中进行第二个测试, 表示“因为break条件而提前退出”。利用你开发的方法, 删除图2.26中的break语句。

- 2.61 下列程序有什么作用?

```

for ( i = 1; i <= 5; i++ ) {
    for ( j = 1; j <= 3; j++ ) {
        for ( k = 1; k <= 4; k++ )
            cout << '*'
        cout << endl;
    }
}

```



```
    }  
    cout << endl;  
}
```

- 2.62 描述如何从程序循环中删除 `continue` 语句并转换成结构化语句。利用你开发的方法，删除图 2.27 中的 `continue` 语句。

**练习 2.64 对应于第 2.22 节“有关对象的思考”。**

- 2.63 用 200 个左右的字描述汽车及其作用，列出其中的名词和动词。本文曾介绍过，每个名词可能对应一个系统中要实现的对象，这里对应的就是汽车系统中要实现的对象。从列出的对象中选择 5 个，对每个对象各列出几个属性和行为，简单描述这些对象如何相互交互并与描述的其他对象交互。这样就完成了典型的面向对象设计的几个关键步骤。
- 2.64 (**Peter Minuit 问题**) 故事发生在 1626 年，Peter Minuit 用 24 美元购买了曼哈顿城 (Manhattan)，这笔投资是否合算呢？要回答这个问题，修改图 2.21 的复利程序，本金为 24.00 美元，要计算从 1626 年到 1998 年 372 年的存款利息，分别用利率 5%、6%、7%、8%、9% 和 10% 计算，看看复利有多大。

## 第3章 函 数

### 教学目标

- 了解如何由函数模块化地构造程序
- 能够生成新函数
- 了解函数之间传递信息的机制
- 介绍使用随机数产生器的模拟技术
- 了解标识符如何限于特定程序区域
- 如何编写和使用调用自己的函数

### 3.1 简介

解决实际问题的大多数程序都比前几章介绍的程序要大得多。经验表明，要开发和维护大程序，最好的办法是从更容易管理的小块和小组件开始。这种方法称为“分而治之，各个击破”(divide and conquer)。本章介绍 C++ 语言中的许多关键特性，可以帮助设计、实现、操作和维护大程序。

### 3.2 C++ 程序组件

C++ 中的模块称为函数 (function) 和类 (class)。C++ 程序一般是将程序员编写的新函数与 C++ 标准库 (standard library) 中提供的预装函数组合而成的，通常是将程序员编写的新类与各种类库中提供的预装类组合而成的。本章主要介绍函数，第 6 章开始将详细介绍类。

C++ 标准库提供了丰富的函数集合，可以进行常用数学计算、字符串操作、字符操作、输入/输出、错误检查和许多其他有用的操作。这就使程序员的工作更加轻松，因为这些函数提供了程序员需要的许多功能。C++ 标准库函数是在 C++ 编程环境中提供的。

#### 编程技巧 3.1

要熟悉 C++ 标准库提供的类和函数集合。

#### 软件工程视点 3.1

不要事事从头做起。要尽可能利用 C++ 标准库提供的函数而不是生成新函数，从而减少程序开发时间。

#### 可移植性提示 3.1

利用 C++ 标准库提供的函数能使程序可移植性更强。

#### 性能提示 3.1

不要改写现有库程序以使其更有效，这些过程的性能通常已经是最优的。

程序员可以通过编写函数定义程序中多处使用的特定任务，这些通常称为程序员定义的函数（programmer-defined function）。定义函数的语句通常只写一次，这些语句是其他函数无法访问的。

通过函数调用（function call）来调用（invoke，即让其完成指定任务）函数。函数调用指定函数名和提供被调用函数完成工作所需的信息（作为参数）。可以把这种形式与管理的层次形式相比，老板（调用函数或调用者）要求工人（被调用函数）完成任务并在任务完成之后返回（即报告）结果。老板函数并不知道工人函数如何完成工作。工人又可能调用其他工人函数，这是老板所不知道的。稍后会介绍这种隐藏实现细节如何促进良好的软件工程。图3.1显示了main函数以层次方式和几个工人函数通信。注意worker1是worker4和worker5的老板函数。函数之间的关系也可能与图中所示的层次结构有所不同。

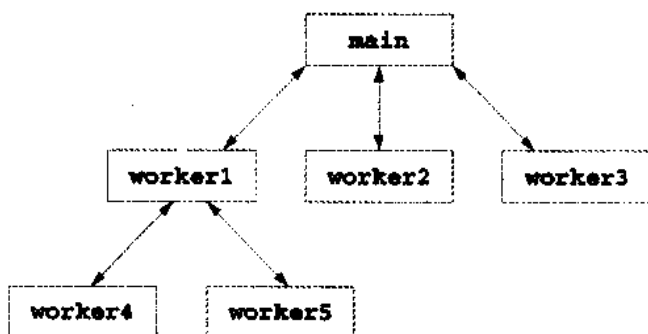


图 3.1 层次化的老板函数 / 工人函数关系

### 3.3 数学函数库

数学函数库使程序员可以进行某些常见数学计算。我们这里用各种数学库函数介绍函数概念。本书稍后会介绍 C++ 标准库中的许多其他函数。

调用函数时，通常写上函数名，然后是一对括号，括号中写上函数参数（或逗号分隔的参数表）。例如程序员可以用下列语句计算和打印 900.0 的平方根：

```
cout << sqrt(900.0);
```

执行这个语句时，数学库函数 sqrt 计算括号中所包含数字（900.0）的平方根。数字 900.0 是 sqrt 函数的参数。上述语句打印 30。sqrt 函数取 double 类型参数，返回 double 类型结果。数学函数库中的所有函数都返回 double 类型结果。要使用数学库函数，需要在程序中包含 math.h 头文件（这个头文件在新的 C++ 标准库中称为 cmath）。

#### 常见编程错误 3.1

使用数学库函数而不包括 math 头文件是个语法错误。程序中使用的每个标准库函数都应包括标准头文件。

函数参数可取常量、变量或表达式。如果 c1 = 13.0、d = 3.0 和 f = 4.0，则下列语句：

```
cout << sqrt (c1 + d * f );
```

计算并打印  $13.0 + 3.0 * 4.0 = 25.0$  的平方根，即 5（因为 C++ 通常对没有小数部分的浮点数不打印小数点和后面的零）。

图 3.2 总结了一些数学库函数。图中变量 x 和 y 为 double 类型。

函数	说明	举例
ceil(x)	将 x 取整为不小于 x 的最小整数	ceil(9.2)=10.0 ceil(-9.8)=-9.0
cos(x)	x (弧度) 的余弦	cos(0.0)=1.0
exp(x)	指数函数 $e^x$	exp(1.0)=2.71828 exp(2.0)=7.38906
fabs(x)	x 的绝对值	x>0, abs(x)=x x=0, abs(x)=0.0 x<0, abs(x)=-x
floor(x)	将 x 取整为不大于 x 的最大整数	floor(9.2)=9.0 floor(-9.8)=-10.0
fmod(x, y)	x/y 的浮点数余数	fmod(13.657, 2.333)=1.992
log(x)	x 的自然对数 (底数为 e)	log(2.71828)=1.0 log(7.389056)=2.0
log10(x)	x 的对数 (底数为 10)	log(10.0)=1.0 log(100.0)=2.0
pow(x, y)	x 的 y 次方 ( $x^y$ )	pow(2, 7)=128 pow(9, .5)=3
sin(x)	x (弧度) 的正弦	sin(0.0)=0
sqrt(x)	x 的平方根	sqrt(900.0)=30.0 sqrt(9.0)=3.0
tan(x)	x (弧度) 的正切	tan(0.0)=0

图 3.2 常用数学库函数

### 3.4 函数

函数使程序员可以将程序模块化。函数定义中声明的所有变量都是局部变量 (local variable), 只在所在的函数中有效。大多数函数有一系列参数, 提供函数之间沟通信息的方式。函数参数也是局部变量。

#### 软件工程视点 3.2

在包含多个函数的程序中, main 应实现为一组函数调用, 这些函数进行大量的程序工作。

将程序函数化的目的有几个, “分而治之、各个击破”的方法使程序开发更容易管理。另一个目的是软件复用性 (software reusability), 用现有函数作为基本组件, 生成新程序。软件复用性是面向对象编程的主要因素。有了好的函数命名和定义, 程序就可以由完成特定任务的标准化函数生成, 而不必用自定义的代码生成。第三个目的是避免程序中的重复代码, 将代码打包成函数使该代码可以从程序中的多个位置执行, 只要调用函数即可。

#### 软件工程视点 3.3

每个函数只限于完成一个定义良好的任务, 函数名应有效地表达这个任务, 这样可以提高软件复用性。

#### 软件工程视点 3.4

如果无法用简单名称表达函数的作用, 则可能是定义的函数要完成的任务太分散。通常应把这种函数分解为几个更小的函数。

## 3.5 函数定义

前面介绍的每个程序都有一个main函数，调用标准库函数完成工作。现在要考虑程序员如何编写自定义函数。

考虑一个程序，用自定义函数square计算整数1到10的平方（如图3.3）。

```
1 // Fig. 3.3: fig03_03.cpp
2 // Creating and using a programmer-defined function
3 #include <iostream.h>
4
5 int square( int );    // function prototype
6
7 int main( )
8 {
9     for ( int x = 1; x <= 10; x++ )
10         cout << square( x ) << " ";
11
12     cout << endl;
13     return 0;
14 }
15
16 // Function definition
17 int square( int y )
18 {
19     return y * y;
20 }
```

输出结果：

1 4 9 16 25 36 49 64 81 100

图 3.3 生成和使用自定义函数

### 编程技巧 3.2

在函数定义之间加上空行，分隔函数和提高程序可读性。

main 中调用函数 square 如下所示：

```
square(x)
```

函数 square 在参数 y 中接收 x 值的副本。然后 square 计算  $y*y$ ，所得结果返回 main 中调用 square 的位置，并显示结果，注意函数调用不改变 x 的值。这个过程用 for 重复结构重复十次。

square 定义表示 square 需要整数参数 y。函数名前面的关键字 int 表示 square 返回一个整数结果。square 中的 return 语句将计算结果返回调用函数。

第 5 行：

```
int square( int );
```

是个函数原型（function prototype）。括号中的数据类型 int 告诉编译器，函数 square 要求调用者提供整数值。函数名 square 左边的数据类型 int 告诉编译器，函数 square 向调用者返回整数值。编译器通过函数原型检查 square 调用是否包含正确的返回类型、参数个数、参数类型和参数顺序。如果函数定义出现在程序中首次使用该函数之前，则不需要函数原型，这种情况下，函数定义也作为函数

原型。如果图 3.3 中第 17 行到第 20 行在 main 之前, 则第 5 行的函数原型是不需要的。函数原型将在第 3.6 节详细介绍。

函数定义格式如下:

```
return-value-type function-name( parameter-list)
{
    declarations and statements
}
```

函数名 (function-name) 是任何有效标识符, 返回值类型 (return-value-type) 是函数向调用者返回的数据类型, 返回值类型 void 表示函数没有返回值。不指定返回值类型时默认为 int。

#### 常见编程错误 3.2

如果函数原型指定返回类型不是 int, 则函数定义中省略返回值类型是个语法错误。

#### 常见编程错误 3.3

需要返回值的函数中不返回值是个语法错误。

#### 常见编程错误 3.4

返回类型声明为 void 的函数中返回值是个语法错误。

#### 编程技巧 3.3

尽管省略返回类型时默认为 int, 但最好显式指定返回类型。

参数表是逗号分隔的清单, 包含函数被调用时接受的参数声明。如果函数不接受任何值, 则参数表为 void 或空白。函数参数表中的每个参数应显式指定类型。

#### 常见编程错误 3.5

同类函数参数应声明为 float x, float y 而不是 float x, y, 参数声明 float x, y 实际上会报告编译错误, 因为参数表中的每个参数应显式指定类型。

#### 常见编程错误 3.6

将分号放在函数定义中参数表的右括号之后是个语法错误。

#### 常见编程错误 3.7

函数中再次把函数参数定义为局部变量是个语法错误。

#### 编程技巧 3.4

向函数传递的参数和函数定义中的对应参数可以同名, 但最好不要同名, 以免引起歧义。

#### 常见编程错误 3.8

函数调用中的() 实际上是 C++ 中的运算符, 使函数可以被调用。如果函数不取参数, 则省略函数调用中的() 并不是语法错误, 但函数可能会在需要的时候不能调用。

花括号中的声明 (declaration) 和语句 (statement) 构成函数体 (function body), 函数体也称为块 (block), 块是包括声明的复合语句。变量可以在任何块中声明, 而且块也可以嵌套。任何情况下不能在一个函数中定义另一个函数。

#### 常见编程错误 3.9

在一个函数中定义另一个函数是个语法错误。

**编程技巧 3.5**

选择有意义的函数名和有意义的参数名能使程序更易读，避免大量使用注释语句。

**软件工程视点 3.5**

函数应能放在一个编辑器窗口中。不管函数有多长，应该都能很好地完成一个任务。小函数能提高函数的复用性。

**软件工程视点 3.6**

程序应写成一组小函数的集合，使得程序更容易编写、调试、维护和修改。

**软件工程视点 3.7**

需要大量参数的函数可能要完成大量的任务。应把函数分成更小的函数来完成各个任务，函数的首部最好能在一行中放下。

**常见编程错误 3.10**

如果函数原型、函数首部与函数调用的形参和实参的个数、类型、顺序以及返回值类型不相符，则是个语法错误。

将控制返回函数调用点的方法有三种。如果函数不返回结果，则控制在到达函数结束的右花括号时或执行下列语句时返回：

```
return;
```

如果函数返回结果，则下列语句：

```
return expression;
```

向调用者返回表达式的值。

第二个例子用自定义函数 maximum 确定和返回三个整数中的最大值（如图 3.4）。输入三个整数，然后将整数传递到 maximum 中，确定最大值。这个值用 maximum 中的 return 语句返回 main。返回的值赋给变量 largest，然后打印。

```
1 // Fig. 3.4: fig03_04.cpp
2 // Finding the maximum of three integers
3 #include <iostream.h>
4
5 int maximum( int, int, int ); // function prototype
6
7 int main( )
8 {
9     int a, b, c;
10
11     cout << "Enter three integers: ";
12     cin >> a >> b >> c;
13
14     // a, b and c below are arguments to
15     // the maximum function call
16     cout << "Maximum is: " << maximum( a, b, c ) << endl;
17
18     return 0;
19 }
20
21 // Function maximum definition
```

```
22 // x, y and z below are parameters to
23 // the maximum function definition
24 int maximum( int x, int y, int z )
25 {
26     int max = x;
27
28     if ( y > max )
29         max = y;
30
31     if ( z > max )
32         max = z;
33
34     return max;
35 }
```

**输出结果:**

```
Enter three integers: 22 85 17
Maximum is: 85
```

```
Enter three integers: 92 35 14
Maximum is: 92
```

```
Enter three integers: 45 19 98
Maximum is: 98
```

图 3.4 自定义函数 maximum

## 3.6 函数原型

C++ 的最重要特性之一是函数原型 (function prototype)。函数原型告诉编译器函数名称、函数返回的数据类型、函数要接收的参数个数、参数类型和参数顺序, 编译器用函数原型验证函数调用。旧版 C 语言不进行这种检查, 因此函数调用出错时, 编译器可能无法发现错误。这种调用可能造成致命运行时错误或非致命运行时错误, 导致很难确认的逻辑错误, 函数原型能纠正这个缺陷。

**软件工程视点 3.8**

C++ 中要求函数原型。用 #include 预处理指令从相应库的头文件中取得标准库函数的函数原型。也可以用 #include 取得包含读者和小组成员使用的函数原型的头文件。

**软件工程视点 3.9**

如果函数定义出现在程序中首次使用函数之前, 则不需要函数原型, 这时函数定义就作为函数原型。

图 3.4 中 maximum 函数原型为:

```
int maximum( int, int, int );
```

这个原型表示 maximum 取三个 int 类型参数, 返回 int 类型结果。注意这个函数原型与 maximum 函数定义的首部相同, 只是不包括参数名 (x、y、z)。

**编程技巧 3.6**

许多程序员用函数原型中的参数名来说明函数, 编译器将忽略这些名称。



**常见编程错误 3.11**

忘记函数原型末尾的分号是个语法错误。

函数原型中包括函数名和参数类型的部分称为函数签名 (function signature) 或签名 (signature)。函数签名不包括函数返回类型。

**常见编程错误 3.12**

函数调用不符合函数原型是个语法错误。

**常见编程错误 3.13**

函数定义不符合函数原型是个语法错误。

作为上述“常见编程错误”的例子，图 3.4 中如果函数原型写成：

```
void maximum( int, int, int );
```

则编译器报告错误，因为函数原型中的 void 返回类型与函数首部中的 int 返回类型不同。

函数原型的另一个重要特性是强制参数类型转换 (coercion of argument)，即强制参数为相应类型。例如，数学库函数 sqrt 可以用整数参数调用，虽然 math.h 中的函数原型指定 double 参数，但函数仍然可以顺利工作。下列语句：

```
cout << sqrt( 4 );
```

能正确地求值 sqrt(4)，并打印结果 2。函数原型使编译器将整数参数值 4 变为 double 值 4.0，然后再传入 sqrt 中。一般来说，与函数原型中参数类型不完全相符的参数值转换为正确类型之后再行函数调用。这种转换可能在不遵照 C++ 提升规则 (promotion rule) 时造成不正确的结果。提升规则指定一种类型转换为另一种类型时怎样才能不丢失数据。在上述 sqrt 的例子中，int 自动转换为 double 而不改变数值，但如果将 double 转换为 int，则会截去 double 的小数部分。将大整数的类型变为小整数的类型 (例如 long 变为 short) 可能造成数值改变。

提升规则适用于包含两种或多种数据类型的表达式，这种表达式也称为混合类型表达式 (mixed-type expression)。混合类型表达式中每个值的类型提升为表达式中最高的类型 (实际上是生成每个值的临时值并在表达式中使用，原值保持不变)。提升的另一个常见用法是在函数参数类型不符合函数定义中指定的参数类型时。图 3.5 显示了内部数据类型由高到低的顺序。

数据类型	
long double	
double	
float	
unsigned long int	(同 unsigned long)
long int	(同 long)
unsigned int	(同 unsigned)
int	
unsigned short int	(同 unsigned short)
short int	(同 short)
unsigned char	
short	
char	

图 3.5 内部数据类型由高到低的顺序

将数值转换为较低类型可能导致数值不正确。因此,要将数值转换为较低类型,只能显式将该值赋给较低类型的变量或使用强制类型转换运算符。函数参数值变为函数原型中的参数类型,就像直接赋给这些类型的变量一样。如果 square 函数使用整型参数(图 3.3)而用浮点参数调用,则该参数换算为 int(将数值转换为较低类型),square 通常返回不正确的值。例如,square(4.5)返回 16 而不是 20.25。

#### 常见编程错误 1.14

将提升规则中较高的数据类型变为较低的数据类型可能改变数据值。

#### 常见编程错误 1.15

函数未定义先调用时如果不定义函数原型属于语法错误。

#### 软件工程视点 3.10

在文件中,放在任何函数定义之外的函数原型可用于该函数原型之后所有对该函数的调用。放在函数之中的函数原型只适用于该函数中的调用。

## 3.7 头文件

每个标准库都有对应的头文件(header file),包含库中所有函数的函数原型和这些函数所需各种数据类型和常量的定义。图 3.6 列出了 C++ 程序中可能包括的常用 C++ 标准库头文件。图 3.6 中多次出现的宏(macro)将在第 17 章“预处理器”中详细介绍。以.h 结尾的头文件是旧式头文件。对每个旧式头文件,我们介绍新标准中使用的版本。

程序员可以生成自定义头文件,自定义头文件应以.h 结尾,可以用#include 预处理指令包括自定义头文件。例如,square.h 头文件可以用下列指令:

```
#include "square.h"
```

放在程序开头。17.2 节介绍了包含头文件的其他信息。

标准库头文件	说明
旧式头文件(本书前面使用)	
<assert.h>	包含增加诊断以帮助程序调试的宏和信息。这个头文件的新版本为<cassert>
<ctype.h>	包含测试某些字符属性的函数原型和将小写字母变为大写字母、将大写字母变为小写字母的函数原型。这个头文件的新版本为<cctype>
<float.h>	包含系统的浮点长度限制。这个头文件的新版本为<cfloat>
<limits.h>	包含系统的整数长度限制。这个头文件的新版本为<climits>
<math.h>	包含数学库函数的函数原型。这个头文件的新版本为<cmath>
<stdio.h>	包含标准输入/输出库函数的函数原型及其使用的信息。这个头文件的新版本为<cstdio>
<stdlib.h>	包含将数字变为文本、将文本变为数字、内存分配、随机数和各种其他工具函数的函数原型,这个头文件的新版本为<cstdlib>
<string.h>	包含 C 语言式的字符串处理函数的函数原型。这个头文件的新版本为<cstring>
<time.h>	包含操作时间和日期的函数原型和类型。这个头文件的新版本为<ctime>
<iostream.h>	包含标准输入/输出函数的函数原型。这个头文件的新版本为<iostream>
<iomanip.h>	包含能够格式化数据流的流操纵算子的函数原型。这个头文件的新版本为<iomanip>
<fstream.h>	包含从磁盘文件输入和输出到磁盘文件的函数的函数原型。这个头文件的新版本为<fstream>

(续表)

标准库头文件	说明
新式头文件 (本书后面使用)	
<utility>	包含许多标准库头文件使用的类和函数
<vector>、<list>、 <deque>、<queue>、 <stack>、<map>、 <set>、<bitset>	包含实现标准库容器的类的头文件。容器在程序执行期间用于存放数据。我们将在“标准模板库”一章介绍这些头文件
<functional>	包含用于标准库算法的类和函数
<memory>	包含用于向标准库容器分配内存的标准库使用的类和函数
<iterator>	包含标准库容器中操作数据的类
<algorithm>	包含标准库容器中操作数据的函数
<exception>	这些头文件包含用于异常处理的类 (见第 13 章)
<stdexcept>	
<string>	包含标准库中 string 类的定义 (见第 19 章)
<sstream>	包含从内存字符串输入和输出到内存字符串的函数原型 (见第 14 章)
<locale>	包含通常在流处理中用于处理其他语言形式数据的类和函数 (例如货币格式、排序字符串、字符表示等等)
<limits>	包含定义每种计算机平台特定数字数据类型的类
<typeinfo>	包含运行时类型信息的类 (在执行时确定数据类型)

图 3.6 常用 C++ 标准库头文件

### 3.8 随机数产生器

下面要介绍一个在模拟事件和游戏的程序中常用的组件。本节和下节开发一个结构良好、包括多个函数的游戏程序。程序中要使用前面介绍的大多数控制结构。

在赌场上，人人都关心的一个问题就是机会元素 (element of chance)，也就是赢钱的运气。这个机会元素可以用标准库中的 rand 函数引入计算机应用程序中。

考虑下列语句：

```
i = rand ( );
```

rand 函数产生 0 到 RAND\_MAX 之间的整数 (这是 <stdlib.h> 头文件中定义的符号常量)。RAND\_MAX 的值至少应为 32767，也就是两个字节 (即 16 位) 所能表示的最大整数值。如果 rand 函数真的可以随机产生整数，则每次调用 rand 函数时，0 到 RAND\_MAX 之间的每个数出现的机会 (chance) 或概率 (probability) 是相等的。

rand 函数产生的数值范围可能与特定应用中所要求的数值范围不同。例如，模拟掷硬币的程序只要 0 (正面) 和 1 (反面)，模拟投骰子的程序只要 1 到 6 之间的随机整数，视频游戏中预测飞船 (有四种类型) 下一个类型的程序只要 1 到 4 之间的随机整数。

要演示 rand 函数，我们开发一个程序，模拟投骰子 20 次并打印每次的值。rand 函数的函数原型见 <stdlib.h>。我们使用求模运算符 (%) 和 rand 函数，如下所示：

```
rand( ) % 6
```

产生 0 到 5 的整数，称为比例缩放 (scaling)。数字 6 称为比例因子 (scaling factor)。然后将产生的数值范围加 1，得到所要结果。图 3.7 确认了产生的结果在 1 到 6 之间。

要显示这些数值的出现机会近似均等，图 3.8 模拟投骰子 6000 次。1 到 6 的每个数值大约都出现 1000 次。

从程序输出可见，通过比例缩放和移动，可以利用 rand 函数真实地模拟投骰子。注意程序用不到 switch 结构中提供的 default case，但我们还是加上 default case，这是个良好的编程习惯。第 4 章数组将介绍如何把整个 switch 结构转换成简单的单行语句。

```

1 // Fig. 3.7: fig03_07.cpp
2 // Shifted, scaled integers produced by 1 + rand( ) % 6
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6
7 int main( )
8 {
9     for ( int i = 1; i <= 20; i++ ) {
10         cout << setw( 10 ) << ( 1 + rand( ) % 6 );
11
12         if ( i % 5 == 0 )
13             cout << endl;
14     }
15
16     return 0;
17 }

```

输出结果：

5	5	3	5	5
2	4	2	5	5
5	3	2	2	1
5	1	4	6	4

图 3.7 比例缩放和移动 “1+rand() % 6” 产生的整数

```

1 // Fig. 3.8: fig03_08.cpp
2 // Roll a six-sided die 6000 times
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6
7 int main( )
8 {
9     int frequency1 = 0, frequency2 = 0,
10         frequency3 = 0, frequency4 = 0,
11         frequency5 = 0, frequency6 = 0,
12         face;
13
14     for ( int roll = 1; roll <= 6000; roll++ ) {
15         face = 1 + rand( ) % 6;
16
17         switch ( face ) {
18             case 1:
19                 ++frequency1;
20                 break;
21             case 2:
22                 ++frequency2;
23                 break;

```

```

24         case 3:
25             ++frequency3;
26             break;
27         case 4:
28             ++frequency4;
29             break;
30         case 5:
31             ++frequency5;
32             break;
33         case 6:
34             ++frequency6;
35             break;
36         default:
37             cout << "should never get here!";
38     }
39 }
40
41 cout << "Face" << setw( 13 ) << "Frequency"
42      << "\n  1" << setw( 13 ) << frequency1
43      << "\n  2" << setw( 13 ) << frequency2
44      << "\n  3" << setw( 13 ) << frequency3
45      << "\n  4" << setw( 13 ) << frequency4
46      << "\n  5" << setw( 13 ) << frequency5
47      << "\n  6" << setw( 13 ) << frequency6 << endl;
48
49 return 0;
50 }

```

**输出结果：**

Face	Frequency
1	987
2	984
3	1029
4	974
5	1004
6	1022

图 3.8 模拟投骰子 6000 次

#### 测试与调试提示 3.1

即使能确切保证程序没有缺陷，也应在 switch 结构中用一个 default case 来找到错误。

再次执行图 3.7 的程序得到：

5	5	3	5	5
2	4	2	5	5
5	3	2	2	1
5	1	4	6	4

注意打印的数值顺序与上一次完全相同，怎么能算是随机数呢？其实，这种可重复性是 rand 函数的一个重要特性。调试程序时，这种可重复性提供了证明修改后程序能正确工作的关键。

rand 函数实际上产生的是伪随机数（pseudo-random number）。重复调用 rand 函数产生一系列看上去是随机的数值，但每次执行程序时，这组数值本身是可重复的。一旦程序进行彻底调试之后，就可以调整为在每次执行程序时产生不同的随机数系列。这个过程称为随机化（randomizing），是

用标准库函数 `srand` 完成的。`srand` 函数取一个 `unsigned` 类型的整数参数并内嵌 `rand` 函数（即种子），就可以在每次执行程序时产生不同的随机数系列。

`srand` 函数的使用如图 3.9 所示。在这个程序中，我们使用数据类型 `unsigned`（`unsigned int` 的缩写）。`int` 值至少占内存的两个字节，可以存正值或负值。`unsigned int` 变量值也至少占内存的两个字节。2 字节的 `unsigned int` 只能取 0 到 65535 的非负值，4 字节的 `unsigned int` 只能取 0 到 4294967295 的非负值，`srand` 函数取 `unsigned int` 值作参数。`srand` 函数的函数原型在头文件 `<stdlib.h>`（新 C++ 标准的 `cstdlib`）中。

下面将程序运行几次并观察结果。注意，只要提供不同的种子，即可在每次执行程序时产生不同的随机数系列。

```
1 // Fig. 3.9: fig03_09.cpp
2 // Randomizing die-rolling program
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6
7 int main( )
8 {
9     unsigned seed;
10
11     cout << "Enter seed: ";
12     cin >> seed;
13     srand( seed );
14
15     for ( int i = 1; i <= 10; i++ ) {
16         cout << setw( 10 ) << 1 + rand( ) % 6;
17
18         if ( i % 5 == 0 )
19             cout << endl;
20     }
21
22     return 0;
23 }
```

**输出结果：**

```
Enter seed: 67
1      6      5      1      4
5      6      3      1      2

Enter seed: 432
4      2      6      4      3
2      5      1      4      4

Enter seed: 67
1      6      5      1      4
5      6      3      1      2
```

图 3.9 投骰子随机化程序

如果不想每次输入种子值而随机化，则要用如下语句：

```
srand( time ( 0 ) );
```

使计算机通过时钟值自动取得种子值。time 函数（上述语句中该函数的参数为 0）返回当前“日历时间”的秒数，将这个值转换为无符号整数值，作为随机数产生器的种子。time 函数的函数原型在 <time.h>（新 C++ 标准的 ctime）中。

#### 性能提示 3.2

srand 函数只要在程序中调用一次即可得到所需的随机化结果，多次调用是多余的，会降低程序性能。

由 rand 函数直接产生的值总是取值为：

$$0 \leq \text{rand}() \leq \text{RAND\_MAX}$$

前面介绍了如何用一句模拟投骰子，该语句如下所示：

```
face = 1 + rand() % 6;
```

总是对变量 face 指定  $1 \leq \text{face} \leq 6$  的整数（随机）。注意这个范围的宽度（即构成该范围的连续整数的个数）为 6 并从 1 开始。从上述语句可以看出，范围宽度是由求模运算符比例缩放 rand 的数值（6）确定的，开始值等于  $\text{rand} \% 6$  中加进的数值（即 1）。可以将这个结果一般化，如下所示：

```
n = a + rand() % b;
```

其中 a 是位移值（等于所要的连续整数范围的开始值），b 是比例因子（即由连续整数构成的该范围的宽度）。练习中将介绍如何从一组非连续整数中随机选择整数。

#### 常见编程错误 3.16

用 srand 函数代替 rand 函数产生随机数是个语法错误，因为 srand 函数不返回值。

### 3.9 案例：机会游戏与 enum 简介

一个最常见的机会游戏是投骰子游戏，游戏规则如下：

游戏者投两枚骰子，每个骰子有六面，这些面包含 1、2、3、4、5、6 个点。投两枚骰子之后，计算点数之和。如果第一次投时的和为 7 或 11，则游戏者获胜。如果第一次投时的和 2、3 或 12，则游戏者输，庄家赢。如果第一次投时的和为 4、5、6、8、9 或 10，则这个和成为游戏者的点数。要想赢，就要继续投骰子，直到赚到点数。如果投七次之后还没有赚到点数，则游戏者输。

图 3.10 中的程序模拟了投骰子游戏，图 3.11 显示了几个示例的执行结果。

```
1 // Fig. 3.10: fig03_10.cpp
2 // Craps
3 #include <iostream.h>
4 #include <stdlib.h>
5 #include <time.h>
6
7 int rollDice( void ); // function prototype
8
9 int main( )
10 {
```

```
11  enum Status { CONTINUE, WON, LOST };
12  int sum, myPoint;
13  Status gameStatus;
14
15  srand( time( NULL ) );
16  sum = rollDice( );           // first roll of the dice
17
18  switch ( sum ) {
19      case 7:
20          case 11:             // win on first roll
21              gameStatus = WON;
22              break;
23      case 2:
24      case 3:
25          case 12:             // lose on first roll
26              gameStatus = LOST;
27              break;
28      default:                 // remember point
29          gameStatus = CONTINUE;
30          myPoint = sum;
31          cout << "Point is " << myPoint << endl;
32          break;               // optional
33  }
34
35  while ( gameStatus == CONTINUE ) {    // keep rolling
36      sum = rollDice( );
37
38      if ( sum == myPoint )             // win by making point
39          gameStatus = WON;
40      else
41          if ( sum == 7 )               // lose by rolling 7
42              gameStatus = LOST;
43  }
44
45  if ( gameStatus == WON )
46      cout << "Player wins" << endl;
47  else
48      cout << "Player loses" << endl;
49
50  return 0;
51 }
52
53 int rollDice( void )
54 {
55     int die1, die2, workSum;
56
57     die1 = 1 + rand( ) % 6;
```



```
58     die2 = 1 + rand( ) % 6;
59     workSum = die1 + die2;
60     cout << "Player rolled " << die1 << " + " << die2
61         << " = " << workSum << endl;
62
63     return workSum;
64 )
```

图 3.10 模拟投骰子游戏的程序

```
Player rolled 6 + 5 = 11
Player wins

Player rolled 6 + 6 = 12
Player loses

Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins

Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

图 3.11 投骰子游戏程序示例的执行结果

注意，游戏者首先要投两枚骰子，后面也是。我们定义rollDice函数投骰子、计算并打印点数和。函数rollDice定义一次，但可从程序的两个地方调用。有趣的是，rollDice不取参数，因此在参数表中用void表示。函数rollDice返回投两枚骰子的点数和，因此在函数首部定义的返回类型为int。

这个游戏相当复杂。游戏者第一次投两枚骰子时可能输也可能赢，也可能投好几次才会定出输赢。变量gameStatus跟踪这个状态，将其声明为Status类型。下列语句：

```
enum Status { CONTINUE, WON, LOST };
```

生成用户自定义类型（user-defined type）即枚举类型（enumeration）。枚举类型由关键字enum和类型名（这里是Status）构成，是一组用标识符表示的整数常量。这些枚举常量（enumeration constant）的值从0开始，增量为1，但也可以指定其他的增量值。在上述枚举中，CONTINUE指定为数值0，WON指定为数值1，LOST指定为数值2。enum中的标识符必须惟一，但不同枚举常量可以取相同的值。

#### 编程技巧 3.7

作为用户自定义类型名的标识符，其第一个字母应大写。

用户自定义类型 `Status` 的变量只能赋给枚举中声明的三个值之一。游戏获胜时, `gameStatus` 设置为 `WON`; 游戏失败时, `gameStatus` 设置为 `LOST`; 否则 `gameStatus` 设置为 `CONTINUE`, 可以再次投骰子。

#### 常见编程错误 3.17

对枚举类型变量指定等价于枚举常量的整数值是个语法错误。

另一个常用枚举是:

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

生成用户自定义类型 `Months`, 用枚举常量表示一年的月份。由于上述枚举中第一个值显式指定为 1, 因此其余值每次递增 1, 取值为 1 到 12。任何枚举常量可以在枚举定义中指定一个整数值, 后面的值用 1 递增。

#### 常见编程错误 3.18

定义枚举常量之后, 想对枚举常量指定另一个值是个语法错误。

#### 编程技巧 3.8

枚举常量名用大写字母能使枚举常量在程序中更醒目, 使程序员注意到枚举常量不是变量。

#### 编程技巧 3.9

用枚举而不用整数常量能使程序更清晰。

第一次投骰子之后, 如果游戏获胜, 则跳过 `while` 结构体, 因为 `gameStatus` 不等于 `CONTINUE`。程序进入 `if/else` 结构, 在 `gameStatus` 等于 `WON` 时打印 “Player wins”, 在 `gameStatus` 等于 `LOST` 时打印 “Player loses”。

第一次投骰子之后, 如果游戏没有结束, 则 `sum` 保存在 `myPoint` 中, 由于 `gameStatus` 等于 `CONTINUE`, 因此执行 `while` 结构中的程序。每次执行 `while` 结构中的程序时, 调用 `rollDice` 产生新的 `sum`。如果 `sum` 符合 `myPoint`, 则 `gameStatus` 设置为 `WON`, `while` 测试失败, `if/else` 结构打印 “Player wins”, 终止执行。如果 `sum` 等于 7, 则 `gameStatus` 设置为 `LOST`, `while` 测试失败, `if/else` 结构打印 “Player loses”, 终止执行。

注意其中使用了前面介绍的各种程序控制机制。投骰子程序使用两个函数 `main` 和 `rollDice`, 并使用 `switch`、`while`、`if/else` 和嵌套 `if` 结构。练习中, 我们要介绍投骰子程序的各种有趣的特点。

## 3.10 存储类

第 1 章到第 3 章用标识符作为变量名。变量属性包括名称、类型、长度和值。本章用标识符作为用户自定义的函数名。实际上, 程序中的每个标识符还有其他属性, 包括存储类 (storage class)、作用域 (scope) 和连接 (linkage)。

C++ 提供了 4 个存储类说明符 (storage class specifier): `auto`、`register`、`extern` 和 `static`。标识符的存储类说明符可以确定其存储类、范围和连接。

标识符的存储类确定了标识符在内存中存在的时间。有些标识符的存在时间很短, 有些则重复生成和删除, 有些存在于整个程序的执行期间。

标识符的作用域是程序中能引用这个标识符的区域。有些标识符可以在整个程序中引用,而有些标识符只能在程序中的有限部分引用。

标识符的连接确定多源文件程序(第6章将会讨论)中,只有当前源文件或是在任何正确声明的源文件中识别标识符。

本节介绍4个存储类说明符和两个存储类。3.11节介绍标识符的作用域。

存储类说明符可以分为两个存储类:自动存储类(automatic storage class)和静态存储类(static storage class)。关键字auto和register用来声明自动存储类变量。这种变量在进入声明的块时生成,在块活动期间存在,在退出这个块时删除。

只有变量能作为自动存储类。函数的局部变量和参数通常是自动存储类。存储类说明符auto显式声明变量为自动存储类。例如,下列声明表示float变量x和y是自动存储类的局部变量,即只在定义该变量的函数体中存在:

```
auto float x, y;
```

局部变量默认为自动存储类,因此关键字auto很少使用。本书余下部分将自动存储类变量简称为自动变量。

#### 性能提示 3.3

自动存储可以节省内存,因为自动存储类变量在进入声明的块时生成并在退出这个块时删除。

#### 软件工程视点 3.11

自动存储是最低权限原则的例子。变量不用时没有必要放在内存中。

机器语言版本中的数据通常装入寄存器(register)中进行计算和其他处理。

#### 性能提示 3.4

存储类说明符register可以放在自动变量声明之前,让编译器在计算机的高速硬件寄存器中而不是内存中保存这个变量。如果能在硬件寄存器中保存计数器、总和等大量使用的变量,则可以消除从内存向寄存器装入变量和将结果返回内存中的重复开销。

#### 常见编程错误 3.19

一个标识符使用多个存储类说明符是个语法错误,一个标识符只能使用一个存储类说明符。例如,如果一个标识符设为register,就不能再设为auto。

编译器可以忽略register声明。例如,编译器可用的寄存器个数可能不足。下列声明建议将counter变量放在计算机的寄存器中,不管编译器是否这么做,counter都初始化为1:

```
register int counter = 1;
```

register关键字只能用于局部变量和函数参数。

#### 性能提示 3.5

register声明通常是不需要的。如今的优化编译器通常能识别经常使用的变量,并决定将其放在寄存器中而不需要程序员进行register声明。

关键字extern和static是用来声明静态存储类变量和函数的标识符。这种变量从程序开始执行时就存在。对于变量,程序开始执行时就分配和初始化存储空间;对于函数,从程序开始执行时就

存在函数名。但是，尽管变量和函数名从程序开始执行时起就存在，但这并不是说这些标识符可以在整个程序中使用。3.11 节将会介绍存储类和作用域是两个不同的概念。

静态存储类有两种标识符：外部标识符（如全局变量和函数名）与存储类说明符 `static` 中声明的局部变量。全局变量和函数名默认为存储类说明符 `extern`。全局变量生成时将变量声明放在任何函数定义之外，在整个程序执行期间保存该全局变量的值。全局变量和函数可以由文件中已声明或定义的任何函数引用。

#### 软件工程视点 3.12

将变量声明为全局变量而不是局部变量可能发生意料不到的副作用，不需要访问该变量的函数可能有意或意外修改这个变量，一般来说，除了有独特性能要求，否则应避免使用全局变量。

#### 软件工程视点 3.13

只在某个函数中使用的变量应声明为该函数中的局部变量，而不是声明为全局变量。

用关键字 `static` 声明的局部变量仍然只在定义该变量的函数中使用，但与自动变量不同的是，`static` 局部变量在函数退出时保持其数值。下次调用这个函数时，`static` 局部变量包含上次函数退出时的值。下列语句将局部变量 `count` 声明为 `static`，并将其初始化为 1。

```
static int count = 1
```

所有静态存储类的数字变量默认初始化为 0，但也可以由程序员显式初始化（第 5 章介绍的静态指针变量也是初始化为 0）。

存储类说明符 `extern` 和 `static` 在显式作用于外部标识符时具有特殊意义。第 18 章将介绍说明符 `extern` 和 `static` 作用于外部标识符和多源文件程序。

## 3.11 作用域规则

程序中一个标识符有意义的部分称为其作用域。例如，块中声明局部变量时，其只能在这个块或这个块嵌套的块中引用。一个标识符的 4 个作用域是函数范围（function scope）、文件范围（file scope）、块范围（block scope）和函数原型范围（function-prototype scope）。后面还要介绍第五个——类范围（class scope）。

任何函数之外声明的标识符取文件范围。这种标识符可以从声明处起到文件末尾的任何函数中访问。全局变量、任何函数之外声明的函数定义和函数原型都取文件范围。

标号（后面带冒号的标识符，如 `start:`）是惟一具有函数范围的标识符。标号可以在所在函数中任何地方使用，但不能在函数体之外引用。标号用于 `switch` 结构中（如 `case` 标号）和 `goto` 语句中（见第 18 章）。标号是函数内部的实现细节，这种信息隐藏（information hiding）是良好软件工程的基本原则之一。

块中声明的标识符的作用域为块范围。块范围从标识符声明开始，到右花括号（`}`）处结束。函数开头声明的局部变量的作用域为块范围，函数参数也是，它们也是函数的局部变量。任何块都可以包含变量声明。块嵌套时，如果外层块中的标识符与内层块中的标识符同名，则外层块中的标识符“隐藏”，直到内层块终止。在内层块中执行时，内层块中的标识符值是本块中定义的，而不是同名的外层标识符值。声明为 `static` 的局部变量尽管在函数执行时就已经存在，但该变量的作用域仍为块范围。存储时间不影响标识符的作用域。

只有函数原型参数表中使用的标识符才具有函数原型范围。前面曾介绍过,函数原型不要求参数表中使用的标识符名称,只要求类型。如果函数原型参数表中使用名称,则编译器忽略这些名称。函数原型中使用的标识符可以在程序中的其他地方复用,不会产生歧义。

#### 常见编程错误 3.20

如果在内层块和外层块中使用同名标识符,而程序员又希望在内层块中引用外层块中的标识符,这通常会产生逻辑错误,因为实际上使用的还是内层块中标识符的值。

#### 编程技巧 3.10

避免隐藏外层块范围名称的变量名,因此要在程序中避免重复使用标识符。

图 3.12 的程序演示了全局变量、自动局部变量和 static 局部变量的作用域问题。

```
1 // Fig. 3.12: fig03_12.cpp
2 // A scoping example
3 #include <iostream.h>
4
5 void a( void );    // function prototype
6 void b( void );    // function prototype
7 void c( void );    // function prototype
8
9 int x = 1;         // global variable
10
11 int main( )
12 {
13     int x = 5;     // local variable to main
14
15     cout << "local x in outer scope of main is " << x << endl;
16
17     {              // start new scope
18         int x = 7;
19
20         cout << "local x in inner scope of main is " << x << endl;
21     }              // end new scope
22
23     cout << "local x in outer scope of main is " << x << endl;
24
25     a( );          // a has automatic local x
26     b( );          // b has static local x
27     c( );          // c uses global x
28     a( );          // a reinitializes automatic local x
29     b( );          // static local x retains its previous value
30     c( );          // global x also retains its value
31
32     cout << "local x in main is " << x << endl;
33
34     return 0;
35 }
36
37 void a( void )
38 {
39     int x = 25;    // initialized each time a is called
40 }
```

```
41  cout << endl << "local x in a is " << x
42      << " after entering a" << endl;
43  ++x;
44  cout << "local x in a is " << x
45      << " before exiting a" << endl;
46 }
47
48 void b( void )
49 {
50     static int x = 50; // Static initialization only
51                        // first time b is called.
52     cout << endl << "local static x is " << x
53         << " on entering b" << endl;
54     ++x;
55     cout << "local static x is " << x
56         << " on exiting b" << endl;
57 }
58
59 void c( void )
60 {
61     cout << endl << "global x is " << x
62         << " on entering c" << endl;
63     x *= 10;
64     cout << "global x is " << x << " on exiting c" << endl;
65 }
```

**输出结果:**

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5
```

```
local x in a is 25 after entering a
local x in a is 26 before exiting a
```

```
local static x is 50 on entering b
local static x is 51 on exiting b
```

```
global x is 1 on entering c
global x is 10 on exiting c
```

```
local x in a is 25 after entering a
local x in a is 26 before exiting a
```

```
local static x is 51 on entering b
local static x is 52 on exiting b
```

```
global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

图 3.12 说明变量作用域的例子

全局变量  $x$  声明并初始化为 1。这个全局变量在任何声明  $x$  变量的块和函数中隐藏。在 `main` 函数中，局部变量  $x$  声明并初始化为 5。打印这个变量，结果表示全局变量  $x$  在 `main` 函数中隐藏。然

后在 main 函数中定义一个新块,将另一个局部变量 x 声明并初始化为 7,打印这个变量,结果表示其隐藏 main 函数外层块中的 x。数值为 7 的变量 x 在退出这个块时自动删除,并打印 main 函数外层块中的局部变量 x,表示其不再隐藏。程序定义三个函数,都没有参数和返回值。函数 a 定义自动变量 x 并将其初始化为 25。调用 a 时,打印该变量,递增其值,并在退出函数之前再次打印该值。每次调用该函数时,自动变量 x 重新初值为 25。函数 b 声明 static 变量 x 并将其初始化为 50。声明为 static 的局部变量在离开作用域时仍然保持其数值。调用 b 时,打印 x,递增其值,并在退出函数之前再次打印该值。下次调用这个函数时,static 局部变量 x 包含数值 51。函数 c 不声明任何变量,因此,函数 c 引用变量 x 时,使用全局变量 x。调用函数 c 时,打印全局变量,将其乘以 10,并在退出函数之前再次打印该值。下次调用函数 c 时,全局变量已变为 10。最后,程序再次打印 main 函数中的局部变量 x,结果表示所有函数调用都没有修改 x 的值,因为函数引用的都是其他范围中的变量。

## 3.12 递归

前面介绍的程序通常由严格按层次方式调用的函数组成。对有些问题,可以用自己调用自己的函数。递归函数 (recursive function) 是直接调用自己或通过另一函数间接调用自己的函数。递归是个重要问题,在高级计算机科学教程中都会详细介绍。本节和下节介绍一些简单递归例子,本书则包含大量递归处理。图 3.17 (3.14 节末尾)总结了本书的递归例子和练习。

我们先介绍递归概念,然后再介绍几个包含递归函数的程序。递归问题的解决方法有许多相同之处。调用递归函数解决问题时,函数实际上只知道如何解决最简单的情况(称为基本情况)。对基本情况的函数调用只是简单地返回一个结果。如果在更复杂的问题中调用函数,则函数将问题分成两个概念性部分:函数中能够处理的部分和函数中不能够处理的部分。为了进行递归,后者要模拟原问题,但稍作简化或缩小。由于这个新问题与原问题相似,因此函数启动(调用)自己的最新副本来处理这个较小的问题,称为递归调用 (recursive call) 或递归步骤 (recursion step)。递归步骤还包括关键字 return,因为其结果与函数中需要处理的部分组合,形成的结果返回原调用者(可能是 main)。递归步骤在原函数调用仍然打开时执行,即原调用还没有完成。

递归步骤可能导致更多递归调用,因为函数继续把函数调用的新的子问题分解为两个概念性部分。要让递归最终停止,每次函数调用时都使问题进一步简化,从而产生越来越小的问题,最终合并到基本情况。这时,函数能识别并处理这个基本情况,并向前一个函数副本返回结果,并回溯一系列结果,直到原函数调用最终把最后结果返回给 main。这一切比起前面介绍的其他问题似乎相当复杂,下面通过一个例子来说明。我们用递归程序进行一个著名的数学计算。

非负整数 n 的阶乘写成  $n!$ , 为下列数的积:

$$n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$$

其中  $1!$  等于 1,  $0!$  定义为 1。例如,  $5!$  为  $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , 即 120。

整数 number 大于或等于 0 时的阶乘可以用下列 for 循环迭代 (非递归) 计算:

```
factorial = 1;
for ( int counter = number; counter >= 1; counter-- )
    factorial *= counter;
```

通过下列关系可以得到阶乘函数的递归定义:

$$n! = n \cdot (n-1)!$$

例如,  $5!$  等于  $5 \cdot 4!$ , 如下所示:

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\ 5! &= 5 \cdot (4!) \end{aligned}$$

求值  $5!$  的过程如图 3.13。图 3.13a) 显示如何递归调用, 直到  $1!$  求值为 1, 递归终止。图 3.13b) 显示每次递归调用向调用者返回的值, 直到计算和返回最后值。

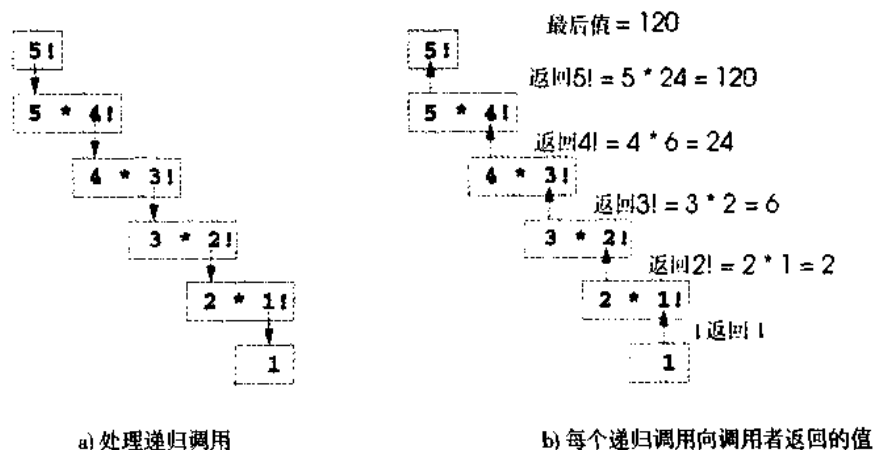


图 3.13 求值  $5!$  的过程

图 3.14 的程序用递归法计算并打印 0 到 10 的整数阶乘 (稍后将介绍数据类型 `unsigned long` 的选择)。递归函数 `factorial` 首先测试终止条件是否为 `true`, 即 `number` 是否小于或等于 1。如果 `number` 小于或等于 1, 则 `factorial` 返回 1, 不再继续递归, 程序终止。如果 `number` 大于 1, 则下列语句。

```
return number * factorial( number - 1 );
```

将问题表示为 `number` 乘以递归调用 `factorial` 求值的 `number-1` 的阶乘。注意 `factorial(number-1)` 比原先 `factorial(number)` 的计算稍微简单一些。

```
1 // Fig. 3.14: fig03_14.cpp
2 // Recursive factorial function
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 unsigned long factorial( unsigned long );
7
8 int main( )
9 {
10     for ( int i = 0; i <= 10; i++ )
11         cout << setw( 2 ) << i << "!" = " << factorial( i ) << endl;
12
13     return 0;
14 }
15
16 // Recursive definition of function factorial
17 unsigned long factorial( unsigned long number )
18 {
```



```
19     if ( number <= 1 )    // base case
20         return 1;
21     else                    // recursive case
22         return number * factorial( number - 1 );
23 }
```

**输出结果：**

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

图 3.14 用递归法计算阶乘

函数 `factorial` 声明为接收 `unsigned long` 类型的参数和返回 `unsigned long` 类型的值。`unsigned long` 是 `unsigned long int` 的缩写，C++ 语言的规则要求存放 `unsigned long int` 类型变量至少占用 4 个字节（32 位），因此可以取 0 到 4294967295 之间的值（数据类型 `long int` 至少占内存中的 4 个字节，可以取  $\pm 2147483647$  之间的值）。如图 3.14，阶乘值很快就变得很大。我们选择 `unsigned long` 数据类型，使程序可以在字长为 16 位的计算机上计算大于 7! 的阶乘。但是，`factorial` 函数很快产生很大的值，即使 `unsigned long` 也只能打印少量阶乘值，然后就会超过 `unsigned long` 变量的长度。

练习中将会介绍，用户最终可能要用 `float` 和 `double` 类型来计算大数的阶乘，这就指出了大多数编程语言的弱点，即不能方便地扩展成处理不同应用程序的特殊要求。从本书面向对象编程部分可以看到，C++ 是个可扩展语言，可以在需要时生成任意大的数。

#### 常见编程错误 3.21

递归函数中需要返回数值时不返回数值将使大多数编译器产生一个警告消息。

#### 常见编程错误 3.22

省略基本情况或将递归步骤错误地写成不能回推到基本情况会导致无穷递归，最终内存用尽。这就像迭代（非递归）解决方案中的无限循环问题。意外输入也可能导致无穷递归。

### 3.13 使用递归举例：Fibonacci 数列

Fibonacci 数列（斐波纳契数列）：

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

以 0 和 1 开头，后续每个 Fibonacci 数是前面两个 Fibonacci 数的和。

自然界中就有这种数列，描述一种螺线形状。相邻 Fibonacci 数的比是一个常量 1.618...，这个数在自然界中经常出现，称为黄金分割（golden ratio 或 golden mean）。人们发现，黄金分割会产生最佳的欣赏效果。因此建筑师通常把窗户、房子和大楼的长和宽的比例设置为黄金分割数，明信片的长宽比也采用黄金分割数。

Fibonacci 数列可以递归定义如下:

```
fibonacci(0)=0
fibonacci(1)=1
fibonacci(n)= fibonacci(n-1)+ fibonacci(n-2)
```

图3.15的程序用函数 `fibonacci` 递归计算第  $i$  个 Fibonacci 数。注意 Fibonacci 数很快也会变得很大, 因此把 `fibonacci` 函数的参数和返回值类型设为 `unsigned long` 数据类型。图3.15中每对输出行显示运行一次程序的结果。

```
1 // Fig. 3.15: fig03_15.cpp
2 // Recursive fibonacci function
3 #include <iostream.h>
4
5 long fibonacci( long );
6
7 int main( )
8 {
9     long result, number;
10
11     cout << "Enter an integer: ";
12     cin >> number;
13     result = fibonacci( number );
14     cout << "Fibonacci(" << number << ") = " << result << endl;
15     return 0;
16 }
17
18 // Recursive definition of function fibonacci
19 long fibonacci( long n )
20 {
21     if ( n == 0 || n == 1 ) // base case
22         return n;
23     else // recursive case
24         return fibonacci( n - 1 ) + fibonacci( n - 2 );
25 }
```

**输出结果:**

```
Enter an integer: 0
Fibonacci(0) = 0
```

```
Enter an integer: 1
Fibonacci(1) = 1
```

```
Enter an integer: 2
Fibonacci(2) = 1
```

```
Enter an integer: 3
Fibonacci(3) = 2
```

```
Enter an integer: 4
Fibonacci(4) = 3
```

```
Enter an integer: 5
Fibonacci(5) = 5
```

```
Enter an integer: 6
Fibonacci(6) = 8
```

```

Enter an integer: 10
Fibonacci(10) = 55
Enter an integer: 20
Fibonacci(20) = 6765
Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465

```

图 3.15 递归计算 Fibonacci 数列

main 中调用 fibonacci 函数不是递归调用, 但后续所有调用 fibonacci 函数都是递归调用。每次调用 fibonacci 时, 它立即测试基本情况 ( $n$  等于 0 或 1)。如果是基本情况, 则返回  $n$ 。有趣的是, 如果  $n$  大于 1, 则递归步骤产生两个递归调用, 各解决原先调用 fibonacci 问题的一个简化问题。图 3.16 显示了 fibonacci 函数如何求值 fibonacci(3), 我们将 fibonacci 缩写成  $f$ 。

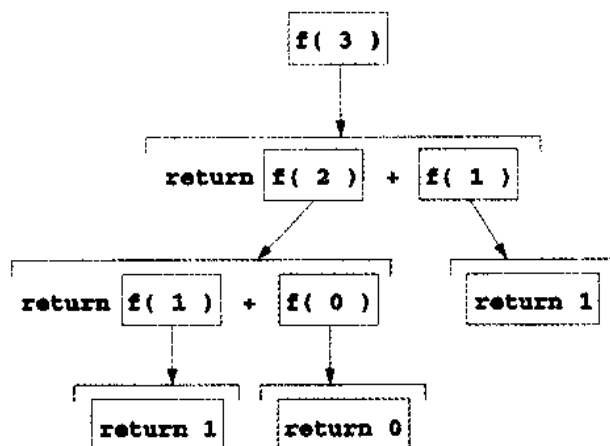


图 3.16 对 fibonacci 数列的递归调用

图中提出了 C++ 编译器对运算符操作数求值时的一些有趣的顺序问题。这个问题与运算符作用于操作数的顺序不同, 后者的顺序是由运算符优先级规则确定的。图 3.16 中显示, 求值  $f(3)$  时, 进行两个递归调用, 即  $f(2)$  和  $f(1)$ 。但这些调用的顺序如何呢?

大多数程序员认为操作数从左向右求值。奇怪的是, C++ 语言没有指定大多数运算符的操作数求值顺序 (包括  $+$ ), 因此, 程序员不能假设这些调用的顺序。调用可能先执行  $f(2)$  再执行  $f(1)$ , 也可能先执行  $f(1)$  再执行  $f(2)$ 。在该程序和大多数其他程序中, 最终结果都是相同的。但在有些程序中, 操作数求值顺序的副作用可能影响表达式的最终结果。

C++ 语言只指定四种运算符的操作数求值顺序, 即 “&&”、“||”、逗号运算符 (,) 和 “?:”。前三种是二元运算符, 操作数从左向右求值。第四种是 C++ 惟一的三元运算符, 先求值最左边的操作数, 如果最左边的操作数为非 0, 则求值中间的操作数, 忽略最后的操作数; 如果最左边的操作数为 0, 则求值最后的操作数, 忽略中间的操作数。

#### 常见编程错误 3.23

对于除 “&&”、“||”、逗号运算符 (,) 和 “?:” 以外的运算符, 如果在编写程序中依赖该运算符操作数的求值顺序, 则会导致错误, 因为编译器不一定按程序员希望的顺序求值操作数。

### 可移植性提示 3.2

对于除“&&”、“||”、逗号运算符(,)和“?:”以外的运算符,如果在程序中依赖该运算符操作数的求值顺序,则不同系统和不同编译器中的结果不同。

要注意这类程序中产生Fibonacci数的顺序。fibonacci函数中的每一层递归对调用数有加倍的效果,即第n个Fibonacci数在第 $2^n$ 次递归调用中计算。计算第20个Fibonacci数就要 $2^{20}$ 次,即上百万次调用,而计算第30个Fibonacci数就要 $2^{30}$ 次,即几十亿次调用。计算机科学家称这种现象为指数复杂性(exponential complexity),这种问题能让最强大的计算机望而生畏。一般复杂性问题 and 指数复杂性将在高级计算机科学课程“算法”中详细介绍。

### 性能提示 3.6

避免fibonacci式的递归程序造成调用的指数性剧增。

## 3.14 递归与迭代

前面几节介绍了两个可以方便地用递归与迭代实现的函数。本节要比较递归与迭代方法,介绍为什么程序员在不同情况下选择不同方法。

递归与迭代都是基于控制结构:迭代用重复结构,而递归用选择结构。递归与迭代都涉及重复:迭代显式使用重复结构,而递归通过重复函数调用实现重复。递归与迭代都涉及终止测试:迭代在循环条件失败时终止,递归在遇到基本情况时终止。使用计数器控制重复的迭代和递归都逐渐到达终止点:迭代一直修改计数器,直到计数器值使循环条件失败;递归不断产生最初问题的简化副本,直到达到基本情况。迭代和递归过程都可以无限进行:如果循环条件测试永远不变成false,则迭代发生无限循环;如果递归永远无法回推到基本情况,则发生无穷递归。

递归有许多缺点,它重复调用机制,因此重复函数调用的开销很大,将占用很长的处理器时间和大量的内存空间。每次递归调用都要生成函数的另一个副本(实际上只是函数变量的另一个副本),从而消耗大量内存空间。迭代通常发生在函数内,因此没有重复调用函数和多余内存赋值的开销。那么,为什么选择递归呢?

### 软件工程视点 3.14

任何能用递归解决的问题也能用迭代(非递归)方法解决。递归方法优于迭代方法之处在于能更自然地反映问题,使程序更容易理解和调试。选择递归方法的另一个原因是可能没有明显的迭代方案。

### 性能提示 3.7

在要求性能的情况下不要用递归方法。递归调用既花时间又占用更多的内存。

### 常见编程错误 3.24

让非递归函数直接或通过另一函数间接调用自己是个逻辑错误。

大多数有关编程的教材都把递归放在后面再讲。我们认为递归问题比较复杂而且内容丰富,应放在前面介绍,本书余下部分会通过更多例子加以说明。图3.17总结了本书使用递归算法的例子和练习。

下面重新考虑书中重复强调的一些观点。良好的软件工程很重要,高性能也很重要,但是,这些目标常常是互相矛盾的。良好的软件工程是使开发的大型复杂的软件系统更容易管理的关键,而高性能是今后在硬件上增加计算需求时实现系统的关键,这两个方面如何取得折衷?

## 软件工程视点 3.15

让程序以整齐有序的层次方式工作能提高软件工程质量，但这是有代价的。

## 性能提示 3.8

功能化的程序（而不是没有函数的一个整体的程序）造成更多的函数调用，需要占用执行时间和计算机处理器空间。但作为一个整体的程序难以编程、测试、调试、维护和改进。

因此要使程序合理地功能化，应该一直保持性能与良好的软件工程之间的平衡。

章号	使用递归算法的例子和练习
第3章	阶乘函数 Fibonacci 函数 最大公约数 两个整数和 两个整数积 一个整数的整数次幂 汉诺塔 逆向打印键盘输入 图形化递归
第4章	数组元素求和 打印数组 逆向打印数组 逆向打印字符串 检查字符串是否为回文 选择数组中的最小值 选择排序 八皇后 线性查找 折半查找
第5章	快速排序 走迷宫 逆向打印键盘输入字符串
第15章	链表插入 链表删除 链表搜索 逆向打印链表 二叉树插入 二叉树前序遍历 二叉树中序遍历 二叉树后序遍历

图 3.17 本书使用递归算法的例子和练习

### 3.15 带空参数表的函数

在 C++ 中，空参数表可以用 void 指定或括号中不放任何东西。下列声明：

```
void print( );
```

指定函数 print 不取任何参数，也不返回任何值。图 3.18 演示了 C++ 声明和使用带空参数表的函数的方法。

**编程技巧 3.11**

虽然函数先定义后使用时可以省略函数原型, 最好也提供函数原型。提供函数原型可以避免代码使用时受到函数定义顺序的限制。(这个顺序可能随程序的演变而改变)。

```
1 // Fig. 3.18: fig03_18.cpp
2 // Functions that take no arguments
3 #include <iostream.h>
4
5 void function1( );
6 void function2( void );
7
8 int main( )
9 {
10     function1( );
11     function2( );
12
13     return 0;
14 }
15
16 void function1( )
17 {
18     cout << "function1 takes no arguments" << endl;
19 }
20
21 void function2( void )
22 {
23     cout << "function2 also takes no arguments" << endl;
24 }
```

**输出结果:**

```
function1 takes no arguments
function2 also takes no arguments
```

图 3.18 两种声明和使用带空参数表函数的方法

**可移植性提示 3.3**

C++ 中带空参数表的函数含义和 C 语言中大不相同。在 C 语言中, 它表示不对所有参数检查 (即函数调用可以传入任何变量), 而在 C++ 中则表示函数不取任何参数。这样, 使用这个特性的 C 语言程序在 C++ 中编译时可能产生语法错误。

介绍省略问题后应该注意, 文件中在任何函数调用之前进行定义的函数不需要另外加上函数原型。这时函数首部就当作函数原型。

**常见编程错误 3.25**

除非提供每个函数的函数原型或先定义再使用每个函数, 否则不能编译 C++ 程序。

## 3.16 内联函数

从软件工程角度看, 将程序实现为一组函数很有好处, 但函数调用却会增加执行时的开销。C++ 提供了内联函数 (inline function) 可以减少函数调用的开销, 特别是对于小函数。函数定义中函数返回类型前面的限定符 inline 指示编译器将函数代码复制到程序中以避免函数调用。其代价是

会产生函数代码的多个副本并分别插入到程序中每一个调用该函数的位置上(从而使程序更大),而不是只有一个函数副本(每次调用函数时将控制传入函数中)。典型情况下,除了最小的函数以外编译器可以忽略用于其他函数的 inline 限定符。

#### 软件工程视点 3.16

任何对内联函数的改变都可能要求函数的所有客户重新编译。这样可能在有些程序的开发和维护中影响非常大。

#### 编程技巧 3.12

inline 限定符只用于经常使用的小函数。

#### 性能提示 3.9

使用内联函数可以减少执行时间,但会增加程序长度。

图 3.19 的程序用内联函数 cube 计算边长为 s 的立方体体积。函数 cube 参数表中的关键字 const 表示函数不修改变量的值 s。关键字 const 将在第 4、5、7 章详细介绍。

```
1 // Fig. 3.19: fig03_19.cpp
2 // Using an inline function to calculate
3 // the volume of a cube.
4 #include <iostream.h>
5
6 inline float cube( const float s ) { return s * s * s; }
7
8 int main( )
9 {
10     cout << "Enter the side length of your cube: ";
11
12     float side;
13
14     cin >> side;
15     cout << "Volume of cube with side "
16           << side << " is " << cube( side ) << endl;
17
18     return 0;
19 }
```

#### 输出结果:

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

图 3.19 用内联函数计算立方体的体积

#### 软件工程视点 3.17

即使被调用函数不修改变量,许多程序员也习惯将数值参数声明为 const。const 只保留原始参数的副本,而不保留原始参数本身。

## 3.17 引用与引用参数

许多编程语言中调用函数的两种方法是按值调用 (call-by-value) 和按引用调用 (call-by-reference)。参数按值调用传递时,生成参数值副本并传给被调用函数。副本的改变并不影响调用者

的原始变量值,这样就可以防止意外的副作用影响开发正确、可靠的软件系统。本章前面的程序中每个传递的参数都是按值调用传递的。

#### 性能提示 3.10

按值调用传递的一个缺点是,如果传递较大的数据项目,则复制这个数据可能要占用相当长的执行时间。

本节介绍引用参数,这是 C++ 提供的两种按引用调用的方法之一。按引用调用时,调用者让被调用函数能够直接访问调用者的数据,并允许被调用函数能够修改其中的数据。

#### 性能提示 3.11

按引用调用对性能有利,因为它消除了复制大量数据的开销。

#### 软件工程视点 3.18

按引用调用的安全性较差,因为被调用函数能够直接访问和修改调用者的数据。

我们将介绍如何利用按引用调用的性能优势同时又满足软件工程的要求(防止破坏调用者数据)。

引用参数是其相应的参数的别名。要表示函数的参数是按引用传递的,只要在函数原型中该参数类型后面加上 & 即可,在函数首部中列出参数类型时也要使用相同的规则。例如,函数首部中的下列声明

```
int &count
```

表示 count 是 int 的引用。在函数调用中,只要指定变量名,该变量就会通过引用传递。在被调用函数体中,通过参数名指定的变量实际上就是引用了调用函数中的原始变量,被调用函数可以直接修改原始变量。一般来讲,函数原型和函数首部必须相符。

图 3.10 比较按值调用、按引用调用与引用参数。调用 squareByValue 和 squareByReference 中,所用参数的形式是相同的,都是只指定名称。如果不检查函数原型或函数定义,要判断被调用函数是否修改了该参数是不可能的。由于函数原型是强制的,因此编译器能够顺利解决歧义性。

```
1 // Fig. 3.20: fig03_20.cpp
2 // Comparing call-by-value and call-by-reference
3 // with references.
4 #include <iostream.h>
5
6 int squareByValue( int );
7 void squareByReference( int & );
8
9 int main( )
10 {
11     int x = 2, z = 4;
12
13     cout << "x = " << x << " before squareByValue\n"
14         << "Value returned by squareByValue: "
15         << squareByValue( x ) << endl
16         << "x = " << x << " after squareByValue\n" << endl;
17
18     cout << "z = " << z << " before squareByReference" << endl;
19     squareByReference( z );
20     cout << "z = " << z << " after squareByReference" << endl;
21
22     return 0;
```



```
23 )  
24  
25 int squareByValue( int a )  
26 {  
27     return a *= a;    // caller's argument not modified  
28 }  
29  
30 void squareByReference( int &cRef )  
31 {  
32     cRef *= cRef;    // caller's argument modified  
33 }
```

**输出结果:**

```
x = 2 before squareByValue  
Value returned by squareByValue: 4  
x = 2 after squareByValue  
  
z = 4 before squareByReference  
z = 16 after squareByReference
```

图 3.20 按引用调用的举例

**常见编程错误 3.26**

由于引用参数在被调用函数体中只指定名称, 因此程序员可能把引用参数当作按值调用的参数。这样, 如果调用函数改变变量原始副本, 则可能产生预想不到的副作用。

第5章介绍指针时, 将会介绍指针提供另一种形式的引用调用, 调用样式能明确表示按引用调用 (可能修改调用者的参数)。

**性能提示 3.12**

如果要传递较大的对象, 用常量引用参数模拟按值调用的情况, 避免传递较大对象副本的开销。

要指定引用常量, 在参数声明的类型说明符前面加上 `const` 限定符。

注意 `squareByReference` 函数参数表中 `&` 的位置。有些 C++ 程序员喜欢写成 `int& cRef` 而不是 `int &cRef`。

**软件工程视点 3.19**

为了获得程序的清晰性和高性能, 许多 C++ 程序员喜欢通过指针将可修改参数传递给函数, 不可修改的小参数按值调用传递, 而不可修改的大参数用常量引用传递给函数。

引用也可以用作函数中其他变量的别名。例如下列代码:

```
int count = 1           // declare integer variable count  
int &cRef = count;      // create cRef as an alias for count  
++cRef;                 // increment count (using its alias)
```

用别名 `cRef` 递增变量 `count` 的值。引用变量应在声明中初始化 (见图 3.21 和图 3.22), 不能作为其他变量的别名而重新赋值。将引用声明为另一变量的别名后, 对该别名 (即引用) 进行的所有操作实际上是对原始变量本身进行的, 别名只是原始变量的另一个名称。获得引用地址和比较引用不会造成语法错误, 而且每个操作实际上是对原始变量本身进行的。引用参数应为左值, 而不能是常量或返回左值的表达式。

**常见编程错误 3.27**

在一条语句中声明多个引用时会出现一些常见问题。例如，要声明  $x$ 、 $y$ 、 $z$  变量为整数的引用，表达式 `int& x=a, y=b, z=c;` 或 `int& x, y, z;` 都是错误的，正确的应是 `int &x=a, &y=b, &z=c;`。

函数可以返回引用，但却会经常出现问题。函数返回被调用函数中声明的变量的引用时，变量应在函数中声明为 `static`，否则引用指的是函数终止时删除的动态变量，这个变量是未定义的，程序的执行情况将无法预测（有些编译器会对此发出警告）。引用未定义变量称为悬挂引用（`dangling reference`）。

**常见编程错误 3.28**

声明引用变量而不对其进行初始化是个语法错误。

```
1 // References must be initialized
2 #include <iostream.h>
3
4 int main( )
5 {
6     int x = 3, &y = x; // y is now an alias for x
7
8     cout << "x = " << x << endl << "y = " << y << endl;
9     y = 7;
10    cout << "x = " << x << endl << "y = " << y << endl;
11
12    return 0;
13 }
```

**输出结果：**

```
x = 3
y = 3
x = 7
y = 7
```

图 3.21 使用初始化的引用

```
1 // References must be initialized
2 #include <iostream.h>
3
4 int main( )
5 {
6     int x = 3, &y; // Error: y must be initialized
7
8     cout << "x = " << x << endl << "y = " << y << endl;
9     y = 7;
10    cout << "x = " << x << endl << "y = " << y << endl;
11
12    return 0;
13 }
```

**输出结果：**

```
Compiling FIG03_21.CPP:
Error FIG03_21.CPP 6:Reference variable 'y' must be
    initialized
```

图 3.22 使用未初始化的引用

**常见编程错误 3.29**

将前面声明的引用重新变为另一变量的别名是个逻辑错误，只是将已经是别名的引用的地址赋给另一变量。

**常见编程错误 3.30**

被调用函数中返回自动变量的指针或引用是个逻辑错误。有些编译器会对此发出警告。

## 3.18 默认参数

函数调用可能通常传递参数的特定值。程序员可以将该参数指定为默认参数，程序员可以提供这个参数的默认值。当函数调用中省略默认参数时，默认参数值自动传递给被调用函数。

默认参数必须是函数参数表中最右边（尾部）的参数。调用具有两个或多个默认参数的函数时，如果省略的参数不是参数表中最右边的参数，则该参数右边的所有参数也应省略。默认参数应在函数名第一次出现时指定，通常是在函数原型中。默认值可以是常量、全局变量或函数调用。默认参数也可以用于内联函数中。

图3.23演示用默认参数计算箱子的容积。第5行boxVolume的函数原型指定所有三个参数的默认值均为1。注意，默认值只能在函数原型中定义，另外，我们在函数原型中提供变量名以增加可读性，当然变量名在函数原型中不是必需的。

```
1 // Fig. 3.23: fig03_23.cpp
2 // Using default arguments
3 #include <iostream.h>
4
5 int boxVolume( int length = 1, int width = 1, int height = 1 );
6
7 int main( )
8 {
9     cout << "The default box volume is: " << boxVolume( )
10         << "\n\nThe volume of a box with length 10,\n"
11         << "width 1 and height 1 is: " << boxVolume( 10 )
12         << "\n\nThe volume of a box with length 10,\n"
13         << "width 5 and height 1 is: " << boxVolume( 10, 5 )
14         << "\n\nThe volume of a box with length 10,\n"
15         << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
16         << endl;
17
18     return 0;
19 }
20
21 // Calculate the volume of a box
22 int boxVolume( int length, int width, int height )
23 {
24     return length * width * height;
25 }
```

**输出结果：**

```
The default box volume is: 1
```

```
The volume of a box with length 10,
width 1 and height 1 is: 10
```

```
The volume of a box with length 10,  
width 5 and height 1 is: 50
```

```
The volume of a box with length 10,  
width 5 and height 2 is: 100
```

图 3.23 用默认参数计算箱子的容积

首次调用函数 `boxVolume` (第 9 行) 时不指定参数, 因此三个参数都用默认值。第二次调用函数 `boxVolume` (第 11 行) 时只传递 `length` 参数, 因此 `width` 和 `height` 参数用默认值。第三次调用函数 `boxVolume` (第 13 行) 时只传递 `length` 和 `width` 参数, 因此 `height` 参数用默认值。最后一次调用函数 `boxVolume` (第 15 行) 时传递 `length`、`width` 和 `height` 参数, 因此不用默认值。

#### 编程技巧 3.13

使用默认值能简化函数调用, 但有些程序员认为显式指定所有参数更清楚。

#### 常见编程错误 3.31

指定和试图使用的默认参数不是最右边的参数 (即没有把该参数右边的参数指定为默认参数)。

## 3.19 一元作用域运算符

可以声明同名的局部变量和全局变量。C++ 提供一元作用域运算符 (`::`), 可以在同名局部变量的作用域中访问全局变量。一元作用域运算符不能在外层块中访问同名的局部变量。如果在作用域内没有与全局变量同名的局部变量, 则可以直接访问全局变量, 而不用一元作用域运算符。第 6 章将介绍类中使用的二元作用域运算符。

图 3.24 演示了当局部变量与全局变量同名时一元作用域运算符的用法。为了突出常量变量 `PI` 的局部和全局版本之间的差别, 程序将其中一个声明为 `double`, 一个声明为 `float`。

#### 常见编程错误 3.32

想用一元作用域运算符在外层块中访问非全局变量时, 如果外层块中没有同名全局变量, 则出现语法错误, 如果有同名全局变量, 则出现逻辑错误。

```
1 // Fig. 3.24: fig03_24.cpp
2 // Using the unary scope resolution operator
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 const double PI = 3.14159265358979;
7
8 int main( )
9 {
10     const float PI = static_cast< float >( ::PI );
11
12     cout << setprecision( 20 )
13         << " Local float value of PI = " << PI
14         << "\nGlobal double value of PI = " << ::PI << endl;
15 }
```

```
16     return 0;
17 }
```

**输出结果：**

```
Local float value of PI = 3.14159
Global double value of PI = 3.14159265358979
```

图 3.24 使用一元作用域运算符

**编程技巧 3.14**

程序中不同用途的变量不要用相同名称。尽管不同用途的变量也可以用相同名称，但容易造成混乱。

## 3.20 函数重载

C++ 允许定义多个同名函数，只要这些函数有不同参数集（至少有不同类型的参数）。这个功能称为函数重载（function overloading）。调用重载函数时，C++ 编译器通过检查调用中的参数个数、类型和顺序来选择相应的函数。函数重载常用于生成几个进行类似任务而处理不同数据类型的同名函数。

**编程技巧 3.15**

用函数重载完成类似的任务可以使程序易于阅读和理解。

图 3.25 用重载函数 square 计算 int 类型值的平方以及 double 类型值的平方。第 8 章将介绍如何重载运算符，定义其如何对用户自定义数据类型的对象进行操作（事实上，我们已经使用了许多已重载的运算符，包括流插入运算符“<<”和流读取运算符“>>”。第 8 章将详细介绍重载“>>”和“<<”）。3.21 节介绍的函数模板自动产生重载函数，对不同数据类型完成相同的任务。第 12 章将详细介绍函数模板和类模板。

```
1 // Fig. 3.25: fig03_25.cpp
2 // Using overloaded functions
3 #include <iostream,h>
4
5 int square( int x ) { return x * x; }
6
7 double square( double y ) { return y * y; }
8
9 int main( )
10 {
11     cout << "The square of integer 7 is " << square( 7 )
12         << "\nThe square of double 7.5 is " << square( 7.5 )
13         << endl;
14
15     return 0;
16 }
```

**输出结果：**

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

图 3.25 使用重载函数

重载函数通过签名 (signature) 进行区别, 签名是函数名和参数类型的组合。编译器用参数个数和类型编码每个函数标识符 (有时称为名字改编或名字修饰), 以保证类型安全连接 (type-safe linkage)。类型安全连接保证调用合适的重载函数并保证形参与实参相符。编译器能探测和报告连接错误。图 3.26 的程序在 Borland C++ 编译器上编译, 我们不显示程序执行的输出, 图中用汇编语言输出了由 Borland C++ 编译器产生的改编函数名。每个改编名用 @ 加上函数名, 改编参数表以 \$q 开头。在函数 nothing2 的参数表中, zc 表示 char、i 表示 int、pf 表示 float\*、pd 表示 double\*。在函数 nothing1 的参数表中, i 表示 int、f 表示 float、zc 表示 char、pi 表示 int\*。两个 square 函数用参数表区分, 一个指定 d 表示 double, 一个指定 i 表示 int。函数的返回类型不在改编名称中指定。函数名改编是编译器特定的。重载函数可以有不同返回类型, 但必须有不同参数表。

#### 常见编程错误 3.33

用不同返回类型和相同参数表生成重载函数是个语法错误。

```
1 // Name mangling
2 int square( int x ) { return x * x; }
3
4 double square( double y ) { return y * y; }
5
6 void nothing1( int a, float b, char c, int *d )
7 { } // empty function body
8
9 char *nothing2( char a, int b, float *c, double *d )
10 { return 0; }
11
12 int main( )
13 {
14     return 0;
15 }
```

#### 输出结果:

```
public _main
public @nothing2$qzqipfpd
public @nothing1$qifzcpd
public @square$qd
public @square$qi
```

图 3.26 名字改编以保证类型安全连接

编译器只用参数表区别同名函数。重载函数不一定要有相同个数的参数。程序员使用带默认参数的重载函数时要小心, 以免出现歧义。

#### 常见编程错误 3.34

调用省略默认参数的函数时可能与调用另一重载函数相同, 这是个语法错误。例如, 如果程序中的函数不带参数, 有一个同名函数包含全部默认参数, 则不带参数调用这个函数时就会造成语法错误。

## 3.21 函数模板

重载函数通常用于不同数据类型用不同程序逻辑进行类似的操作。如果每种数据类型的程序逻辑和操作相同, 那么用函数模板 (function template) 完成这项工作更加简洁和方便。程序员编写一

个函数模板定义。根据函数调用中提供的参数类型, C++ 自动产生不同模板函数 (template function) 来处理不同类型的调用。这样, 定义一个函数模板即可定义一系列的解决方案。

所有的函数模板定义都是以关键字 `template` 开头, 之后是用尖括号 (`<>`) 括起来的形式参数表。每一个形式参数之前都有关键字 `class`。形式参数是内部类型或用户自定义类型, 指定函数的参数类型及返回类型, 在函数定义体中声明变量, 然后和任何其他函数中一样进行函数定义。

图 3.27 使用如下的函数模板定义:

```
template < class T >
T maximum( T value1, T value2, T value3 )
{
    T max = value1;

    if ( value2 > max )
        max = value2;

    if ( value3 > max )
        max = value3;

    return max;
}
```

这个函数模板声明一个形式参数 `T` 为函数 `maximum` 要测试的数据类型。编译器发现程序源代码中的 `maximum` 调用时, 传入 `maximum` 的数据类型代替整个模板定义中的 `T`, C++ 生成一个完整函数, 确定三个指定数据类型值的最大值, 然后编译新生成的函数。可以看出, 模板就是一种代码产生工具。图 3.27 中实例化三个函数: 一个取三个 `int` 值, 一个取三个 `double` 值, 一个取三个 `char` 值。`int` 类型的实例化如下所示:

```
int maximum(int value1, int value2, int value3 )
{
    int max = value1;

    if ( value2 > max )
        max = value2;

    if ( value3 > max )
        max = value3;

    return max;
}
```

模板定义中的每种参数都要在函数的参数表中至少出现一次。类型参数名在特定模板定义的形式参数表中必须惟一。

图 3.27 演示用 `maximum` 确定三个 `int` 值、三个 `double` 值和三个 `char` 值的最大值。

```
1 // Fig. 3.27: fig03_27.cpp
2 // Using a function template
3 #include <iostream.h>
4
5 template < class T >
6 T maximum( T value1, T value2, T value3 )
7 {
8     T max = value1;
9
10    if ( value2 > max )
```

```

11     max = value2;
12
13     if ( value3 > max )
14         max = value3;
15
16     return max;
17 }
18
19 int main( )
20 {
21     int int1, int2, int3;
22
23     cout << "Input three integer values: ";
24     cin >> int1 >> int2 >> int3;
25     cout << "The maximum integer value is: "
26         << maximum( int1, int2, int3 );           // int version
27
28     double double1, double2, double3;
29
30     cout << "\nInput three double values: ";
31     cin >> double1 >> double2 >> double3;
32     cout << "The maximum double value is: "
33         << maximum( double1, double2, double3 ); // double version
34
35     char char1, char2, char3;
36
37     cout << "\nInput three characters: ";
38     cin >> char1 >> char2 >> char3;
39     cout << "The maximum character value is: "
40         << maximum( char1, char2, char3 )         // char version
41         << endl;
42
43     return 0;
44 }

```

#### 输出结果:

```

Input three integer values: 1 2 3
The maximum integer value is: 3
Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3
Input three characters: A C B
The maximum character value is: C

```

图 3.27 使用函数模板

#### 常见编程错误 3.35

函数模板中的每个类型参数前都要放上 `class` 关键字, 否则属于语法错误。

#### 常见编程错误 3.36

在函数签名中不使用函数模板的每个类型参数属于语法错误。

## 3.22 有关对象的思考: 确定对象属性

第2章“有关对象的思考”一节开始了关于电梯模拟程序的面向对象设计(OOD)的第一个阶



段,即确定实现电梯模拟程序所需要的对象。作为入手点,可以列出问题陈述中的名词,这样就可以发现电梯模拟程序中的对象有电梯、人、楼层、大楼、各种按钮、时钟、电灯与电铃等等。

第1章介绍对象时曾经指出过对象有属性和行为。对象属性在C++程序中表示为数据,对象行为在C++程序中表示为函数。本节要确定实现电梯模拟程序所需对象的属性。第4章要确定对象行为。第5章要确定电梯模拟程序中对象与对象之间的交互,称为协作(collaboration)。

下面先介绍现实世界中对象的属性。一个人的属性包括身高、体重等。收音机的属性包括调频、调幅、调台和当前音量的设置。汽车的属性包括车速表和全程表、汽油用量表、档位等等。房子的属性包括建筑风格(古典式、现代式)、房间数、面积和层数。个人计算机的属性包括厂家(Apple、IBM、Compaq等等)、显示器类型(黑白或彩色)、主存大小、硬盘容量等等。

### 电梯实验室任务2

1. 首先在字处理器和编辑器程序中输入模拟电梯的问题陈述文本(见2.22节)。
2. 找出问题中的事实。消除不相关的文本,并将每个事实放在一行(问题陈述中大约有60个事实)。下面是事实文件的第一个部分:

#### 事实文件

两层办公楼

电梯

电梯上的人

门

第1层

方向——上和下

时钟

时间0

每秒滴答一次

电梯模拟程序的调度器组件

随机设置每层第一个到达的人

第一次到达的时间

电梯模拟程序

对指定层生成新的人

将人放在该层

人按下该层的向上或向下按钮

该层的向上或向下按钮

人的目标层

人所在的层

第1层第一个出现的人

出现在第一层的人

人上电梯

人按向上按钮

3. 将所有事实分类组合以确定第2章练习中的类。用概述形式,在页的左边列出类,并缩排一个制表符大小以列出该类相关的事实。有些事实只涉及一个类,有些事实涉及多个类。每个事实最初应列在每个涉及该事实的类中。注意“方向——上和下”等事实并不显式地属于某

个类，但还是要放在一个类中（本例中，该方向是电梯移动的方向）。这个概述文件用于该任务和下面几个任务。

4. 现在将每个类的事实分为两组。第一组标为“属性”，第二组标为“其他事实”。目前操作（行为）应放在“其他事实”中。将操作放在其他事实中时，考虑是否要生成“属性”中的附加项目。例如“电梯关门”是其他事实中的操作，但表示门的属性可开可关。“该层已有人占用”是层的属性，一个层总是被占用或不被占用。电梯的属性包括：运动或停止，有人或无人，向上或向下；按钮的属性是开或关；人的属性是目标层等等。

### 说明

1. 首先列出问题陈述中明确提到的类属性，然后列出问题陈述中隐含的类属性。
2. 在需要时增加相应的属性。
3. 系统设计不是完善和完整的过程，只要尽力而为，后面几章的练习将会介绍如何对设计进行修改。
4. 对象可以是一个类的属性，称为复合。例如，电梯中有第1层和第2层的按钮对象，人需要按这些按钮以选择目标层。本练习中分别处理所有的类，而不进行复合。第5章将介绍复合。
5. 本章介绍了如何实现“随机性”，最终实现电梯模拟程序时，可以用下列语句计划一层中下一个到来的人：

```
arrivalTime = currentTime + ( 5 + rand( ) % 16 );
```

### 小结

- 要开发和维护大程序，最好的办法是从更容易管理的小块和小组件开始。C++ 中的模块称为函数和类。
- 通过函数调用来调用函数。函数调用指定函数名和提供被调用函数完成任务所需的信息（作为参数）。
- 信息隐藏的目的在于函数只能访问完成任务所需要的信息，从而实现最低权限。这是良好的软件的最重要特性之一。
- 调用函数的格式为：先写上函数名，后面跟着左括号，然后是函数参数（或逗号分隔的参数表），最后是右括号。
- double 和 float 都是浮点数据类型。float 类型变量能存放比 double 更大、精度更高的数值。
- 函数参数可取常量、变量或表达式。
- 局部变量只在声明该变量的函数中有效。函数不能知道任何其他函数的实现细节（包括局部变量）。
- 函数定义格式如下：

```
return-value-type function-name( parameter-list)
{
    declarations and statements
}
```

函数名是任何有效标识符，返回值类型是函数向调用者返回的值的的数据类型，返回值类型为 void 表示函数没有返回值。参数表是逗号分隔的清单，包含传递给函数的参数的声明。如果函数不接受任何值，则参数表为 void 或空着。花括号中的声明和语句构成函数体。

- 函数定义和函数调用的形参和实参的个数、类型、顺序和返回值类型应该相符。
- 程序遇到函数时，控制从调用点转入被调用函数，执行该函数，然后控制返回调用者。

- 将控制返回函数调用点的方法有三种。如果函数不返回结果，则控制在到达函数结束的右花括号时或执行下列语句时返回：

```
return;
```

如果函数返回结果，则下列语句：

```
return expression;
```

向调用者返回表达式的值。

- 函数原型声明函数名称、函数返回的数据类型、函数要接收的参数个数、参数类型和参数顺序。
- 编译器用函数原型确定正确地调用了函数。
- 编译器忽略函数原型中提到的参数。
- 每个标准库都有对应的头文件，包含库中所有函数的函数原型和这些函数所需各种数据类型和常量的定义。
- 程序员可以生成和包括自定义的头文件。
- 通过按值调用传递参数时，参数值生成副本并传入被调用函数。副本的改变并不影响调用者的原始变量值。
- rand 函数产生 0 到 RAND\_MAX 之间的整数，RAND\_MAX 的值至少应为 32767。
- rand 和 srand 的函数原型在 <stdlib.h> 中。
- 通过比例缩放和移动，可以调整  $1 + \text{rand}() \% 6$  产生的整数值范围。
- 随机化是用标准库函数 srand 函数完成的。
- srand 语句只在程序完全调试之后才插入程序中。调试时最好省略 srand 语句，这样可以保证重复性，也是保证修改后的随机数产生程序正确工作的基础。
- 如果不想每次输入种子值而随机化，则要用如下语句 `srand(time(0))`，使计算机通过时钟值自动取得种子值。time 函数（上述语句中的参数 0）返回当前“日历时间”的秒数。time 函数的函数原型在 <time.h> 中。
- 可以将随机数结果一般化为如下形式：

```
n = a + rand() % b;
```

其中 a 是位移值（等于所要的连续整数范围的开始值），b 是比例因子（等于由连续整数构成的该范围的宽度）。

- 枚举类型由关键字 enum 和类型名构成，是一组用户标识符表示的整数常量。
- 这些枚举常量的值从 0 开始，增量为 1，但也可以指定其他值。
- enum 中的标识符必须惟一，但不同枚举常量可以取相同的值。
- 任何枚举常量可以在枚举定义中显式地赋给一个整数值。
- 每个标识符都包括存储类、作用域和连接属性。
- C++ 提供了 4 个存储类说明符：auto、register、extern 和 static。
- 标识符的存储类确定标识符在内存中存在的时间。
- 标识符的作用域是程序中能引用这个标识符的区域。
- 标识符的连接确定多源文件程序中，只有当前源文件或是在任何正确声明的源文件中识别标识符。

- 自动存储类变量在进入声明该变量的程序块时生成，在程序块活动期间存在，在退出这个程序块时删除。函数的局部变量和参数通常是自动存储类。
- 存储类说明符 `register` 可以放在自动变量声明之前，让编译器在计算机的高速寄存器中保存这个变量。编译器可以忽略 `register` 声明。关键字 `register` 只能用于自动存储类的变量。
- 关键字 `extern` 和 `static` 声明静态存储类函数和变量的标识符。
- 程序开始执行时就分配和初始化静态存储类。
- 静态存储类有两种标识符：外部标识符与存储类说明符 `static` 中声明的局部变量。
- 全局变量生成时将变量声明放在任何函数定义之外，全局变量在整个程序执行期间保留其数值。
- `static` 局部变量在函数退出时保持其数值。
- 所有静态存储类的数字变量默认初始化为 0，但也可以由程序员显式初始化。
- 一个标识符的 4 个作用域是函数范围、文件范围、块范围和函数原型范围。
- 标号是惟一具有函数范围的标识符。标号可以在所在函数中任何地方使用，但不能在函数体之外引用。
- 任何函数之外声明的标识符的作用域是文件范围。可以从声明该标识符处到文件末尾的任何函数中访问该标识符。
- 块中声明的标识符的作用域是块范围。块范围从标识符声明开始，到表示程序终止的右花括号 `{}` 处结束。
- 函数开头声明的局部变量的作用域是块范围，函数参数也是，它们也是函数的局部变量。
- 任何块都可以包含变量声明。块嵌套时，如果外层块中的标识符与内层块中的标识符同名，则外层块中的标识符“隐藏”，直到内层块终止。
- 只有函数原型参数表中使用的标识符才具有函数原型范围。参数表中使用的标识符可以在程序中其他地方复用，不会产生歧义。
- 递归函数是直接调用自己或通过另一函数间接调用自己的函数。
- 如果函数调用基本情况，则只是返回一个值。如果在更复杂的问题中调用函数，则函数将问题分成两个概念性部分：函数中能够处理的部分和函数中不能够处理的部分。由于这个新问题与原问题相似，因此函数运行递归调用处理这个较小的问题。
- 要让递归最终停止，每次函数调用时都要使问题进一步简化，从而产生越来越小的问题，最终合并到基本情况。这时，函数能识别这个基本情况，将结果返回前一个函数调用，并回溯一系列结果，直到原函数调用最终返回最后的结果。
- C++ 语言只指定四种运算符的操作数求值顺序，即 `&&`、`||`、逗号运算符 `(,)` 和 `?:`。前三种是二元运算符，操作数从左向右求值。第四种是 C++ 惟一的三元运算符，先求值最左边的操作数，如果最左边的操作数非 0，则求值中间的操作数，忽略最后的操作数；如果最左边的操作数为 0，则求值最后的操作数，忽略中间的操作数。
- 递归与迭代都是基于控制结构：迭代用重复结构，而递归用选择结构。
- 递归与迭代都涉及重复操作：迭代显式使用重复结构，而递归通过重复函数调用实现重复。
- 递归与迭代都涉及终止测试：迭代在循环条件失败时终止，递归在遇到基本情况时终止。
- 迭代和递归都可以无限进行：如果循环测试永远不变成 `false`，则迭代发生无限循环；如果递归永远无法归并到基本情况，则发生无穷递归。
- 递归重复调用机制，因此重复函数调用的开销很大，将占用很长的处理器时间和大量的内存空间。

- 如果要编译 C++ 程序, 就要求提供每个函数的函数原型或先定义再使用每个函数。
- 不返回数值的函数用 `void` 返回类型声明。想从这个函数返回一个值或在调用函数中使用这个函数调用的结果是个语法错误。
- 空参数表用空括号或 `void` 指定。
- 内联函数可以减少函数调用开销。关键字 `inline` 指示编译器在适当的时候将函数代码复制到程序中, 以减少函数调用。编译器可以忽略内联函数的声明。
- 引用参数是 C++ 提供的两种按引用调用的方法之一。要表示函数参数按引用传递, 只要在函数原型中参数类型后面加上 `&`。在函数调用中, 只要指定变量名, 即按引用调用传递该变量。在被调用函数中, 通过参数名指定的变量实际上就是引用了调用函数中的原始变量, 被调函数可以直接修改原始变量。
- 引用也可以用作函数中其他变量的别名。引用变量必须在声明中初始化, 不能作为其他变量的别名而重新赋值。将引用声明为另一变量的别名后, 对该别名 (即引用) 进行的所有操作实际上是对原始变量本身进行。
- 程序员可以提供默认参数的默认值。调用函数时如果省略默认参数, 则采用该参数的默认值。如果省略的参数不是参数表中最右边的参数, 则该参数右边的所有参数也应省略。默认参数应在函数名第一次出现时指定, 默认值可以是常量、全局变量或函数调用。
- 一元作用域运算符可以在同名局部变量作用域中访问全局变量。
- C++ 允许定义多个同名函数, 这些函数采用不同参数集 (至少有不同类型的参数)。这个功能称为函数重载。调用重载函数时, C++ 编译器通过检查调用中的参数个数、类型和顺序来选择相应的函数。
- 重载函数可以有不同的返回类型, 但必须有不同参数表。生成不同返回类型和相同参数表的重载函数是个语法错误。
- 函数模板只定义一次, 即可对不同类型数据生成进行相同操作的函数。

## 术语

ampersand (&) suffix	& 号后缀	caller	调用者
argument in a function call	函数调用中的参数	calling function	调用函数
auto storage class specifier	auto 存储类说明符	coercion of arguments	强制参数类型转换
automatic storage	自动存储	collaboration	协作
automatic storage class	自动存储类	component	组件
automatic variable	自动变量	const	
base case in recursion	递归中的基本情况	constant variable	常量变量
block	块	copy of a value	数值副本
block scope	块范围	dangling reference	悬挂引用
C++ standard library	C++ 标准库	default function arguments	默认函数参数
call a function	调用函数	divide and conquer	分而治之, 各个击破
call-by-reference	按引用调用	element of chance	机会元素
call-by-value	按值调用	enum	
called function	被调用函数	enumeration	枚举

enumeration constant 枚举常量	rand
extern storage class specifier extern 存储类说明符	random number generation 随机数产生
factorial function 阶乘函数	randomize 随机化
file scope 文件范围	RAND_MAX
function 函数	read-only variable 只读变量
function call 函数调用	recursion 递归
function declaration 函数声明	recursive call 递归调用
function definition 函数定义	recursive function 递归函数
function overloading 函数重载	reference parameter 引用参数
function prototype 函数原型	reference type 引用类型
function scope 函数范围	register storage class specifier register 存储类说明符
function signature 函数签名	return
global variable 全局变量	return-value-type 返回值类型
header file 头文件	scaling 比例缩放
infinite recursion 无穷递归	scope 作用域 (范围)
information hiding 信息隐藏	shifting 位移
inline function 内联函数	side effect 副作用
invoke a function 调用函数	signature 签名
iteration 迭代	simulation 模拟
linkage 连接	software engineering 软件工程
linkage specification 连接指定	software reusability 软件复用
local variable 局部变量	srand
math library functions 数学库函数	standard library header files 标准库头文件
mixed-type expression 混合类型表达式	static storage class specifier static 存储类说明符
modular program 模块化程序	static storage duration 静态存储期
name decoration 名字修饰	static variable static 变量
name mangling 名字改编	storage class specifier 存储类说明符
named constant 命名常量	storage class 存储类
optimizing compiler 优化编译器	template
overloading 重载	template function 模板函数
parameter in a function definition 函数定义中的参数	time
principle of least privilege 最低权限原则	type-safe linkage 类型安全连接
programmer-defined function 程序员定义的函数	unary scope resolution operator :: 一元作用域运算符
promotion hierarchy 层次提升	unsigned
	void

## 自测练习

### 3.1 填空:

a) C++ 中的程序组件称为 \_\_\_\_\_ 和 \_\_\_\_\_。

- b) 通过 \_\_\_\_\_ 调用函数。
  - c) 只在函数中定义和访问的变量称为 \_\_\_\_\_。
  - d) 被调函数中的 \_\_\_\_\_ 语句将表达式返回调用函数。
  - e) 函数首部用关键字 \_\_\_\_\_ 表示函数不返回值或函数不包含参数。
  - f) 标识符的 \_\_\_\_\_ 是程序中使用该标识符的部分。
  - g) 将控制从被调函数返回调用者的三种方法是 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
  - h) \_\_\_\_\_ 使编译器可以检查传入函数的参数个数、类型和顺序。
  - i) \_\_\_\_\_ 函数用于产生随机数。
  - j) \_\_\_\_\_ 函数设置将程序随机化的随机数种子。
  - k) 存储类说明符有 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
  - l) 函数块或参数表中声明的变量默认为存储类 \_\_\_\_\_，除非另外指定。
  - m) 存储类说明符 \_\_\_\_\_ 建议编译器将变量存放在计算机的寄存器中。
  - n) 块和函数之外声明的变量是 \_\_\_\_\_ 变量。
  - o) 要让函数中的局部变量在函数调用之间保持其数值，则要用存储类说明符 \_\_\_\_\_ 声明。
  - p) 标识符的四种作用域是 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
  - q) 直接或间接调用自己的函数是 \_\_\_\_\_ 函数。
  - r) 递归函数通常有两个组件：一个提供测试 \_\_\_\_\_ 情况以终止递归的方法；另一个将问题表示为比原问题简化的问题。
  - s) 在 C++ 中，可以有多个同名而处理不同参数类型或个数的函数，称为函数 \_\_\_\_\_。
  - t) \_\_\_\_\_ 可以在局部变量范围中访问与局部变量同名的全局变量。
  - u) \_\_\_\_\_ 限定符声明只读变量。
  - v) 函数 \_\_\_\_\_ 使一个函数可以定义成对许多不同数据类型完成同一个任务。
- 3.2 对于如下程序，指出下列元素的作用域（函数范围、文件范围、块范围或函数原型范围）：
- a) main 中的变量 x。
  - b) cube 中的变量 y。
  - c) 函数 cube。
  - d) 函数 main。
  - e) cube 的函数原型。
  - f) cube 函数原型中的标识符 y。

```
1 // ex03_02.cpp
2 #include <iostream.h>
3
4 int cube ( int y );
5
6 int main( )
7 {
8     int x;
9
10    for ( x = 1; x <= 10; x++ )
11        cout << cube (x) << endl;
12
13    return 0;
14 }
15
```

```

16 int cube( int y )
17 {
18     return y * y * y;
19 }

```

3.3 编写一个程序，测试图 3.2 所示数学库函数调用的例子是否产生图中所示结果。

3.4 指定下列函数的函数首部：

- a) 函数 `hypotenuse` 取两个双精度浮点数参数 `side1` 和 `side2`，返回一个双精度浮点数结果。
- b) 函数 `smallest` 取三个整数 `x`、`y`、`z` 并返回一个整数。
- c) 函数 `instructions` 不取参数也不返回数值（注意：这种函数通常用于向用户显示指令）。
- d) 函数 `intToFloat` 取整数参数 `number`，返回浮点数结果。

3.5 指定下列函数原型：

- a) 练习 3.4a) 的函数。
- b) 练习 3.4b) 的函数。
- c) 练习 3.4c) 的函数。
- d) 练习 3.4d) 的函数。

3.6 编写下列声明：

- a) 整数 `count`，放在寄存器中，将 `count` 初始化为 0。
- b) 浮点变量 `lastVal`，在所定义函数调用之间保持其数值。
- c) 外部变量 `number`，其作用域限于所在文件。

3.7 寻找下列程序中的错误并说明如何纠正（也可见练习 3.53）：

```

a) int g( void ) {
    cout << "Inside function g" << endl;
    int h( void ) {
        cout << "Inside function h" << endl;
    }
}

b) int sum( int x, int y ) {
    int result;

    result = x + y;
}

c) int sum( int n ) {
    if ( n == 0 )
        return 0;
    else
        n + sum( n - 1 );
}

d) void f( float a ); {
    float a;

    cout << a << endl;
}

e) void product( void ) {
    int a, b, c, result;
    cout << "Enter three integers: ";
    cin >> a >> b >> c;
    result = a * b * c;
    cout << "Result is " << result;
    return result;
}

```



- 3.8 为什么函数原型包含“float&”之类的参数类型声明?
- 3.9 (判断对错) C++ 中的所有调用都是传值调用。
- 3.10 编写一个完整的 C++ 程序, 用内联函数 sphereVolume 提示用户输入球半径, 计算和打印球的体积, 公式为  $\text{volume} = (4/3) * 3.14159 * \text{pow}(\text{radius}, 3)$ 。

### 自测练习答案

- 3.1 a)函数、类。b)函数调用。c)局部变量。d)return。e)void。f)作用域。g)return; 或 return expression; 或遇到函数结束符号。h)函数原型。i)rand。j)srand。k)auto、register、extern、static。l)auto。m)register。n)外部(全局)。o)static。p)函数范围、文件范围、块范围或函数原型范围。q)递归。r)基本。s)重载。t)一元作用域运算符(::)。u)const。v)模板。
- 3.2 a)块范围。b)块范围。c)文件范围。d)文件范围。e)文件范围。f)函数原型范围。
- 3.3 如下所示:

```

1 // ex03_03.cpp
2 /* Testing the math library functions */
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <math.h>
6
7 int main( )
8 {
9     cout << setiosflags( ios::fixed | ios::showpoint )
10         << setprecision( 1 )
11         << "sqrt(" << 900.0 << ") = " << sqrt( 900.0 )
12         << "\nsqrt(" << 9.0 << ") = " << sqrt( 9.0 )
13         << "\nexp(" << 1.0 << ") = " << setprecision( 6 )
14         << exp( 1.0 ) << "\nexp(" << setprecision( 1 ) << 2.0
15         << ") = " << setprecision( 6 ) << exp( 2.0 )
16         << "\nlog(" << 2.718282 << ") = " << setprecision( 1 )
17         << log( 2.718282 ) << "\nlog(" << setprecision( 6 )
18         << 7.389056 << ") = " << setprecision( 1 )
19         << log( 7.389056 ) << endl;
20     cout << "log10(" << 1.0 << ") = " << log10( 1.0 )
21         << "\nlog10(" << 10.0 << ") = " << log10( 10.0 )
22         << "\nlog10(" << 100.0 << ") = " << log10( 100.0 )
23         << "\nfabs(" << 13.5 << ") = " << fabs( 13.5 )
24         << "\nfabs(" << 0.0 << ") = " << fabs( 0.0 )
25         << "\nfabs(" << -13.5 << ") = " << fabs( -13.5 ) << endl;
26     cout << "ceil(" << 9.2 << ") = " << ceil( 9.2 )
27         << "\nceil (" << -9.8 << ") = " << ceil( -9.8 )
28         << "\nfloor (" << -9.2 << ") = " << floor( -9.2 )
29         << "\nfloor (" << -9.8 << ") = " << floor( -9.8 ) << endl;
30     cout << "pow(" << 2.0 << ", " << 7.0 << ") = "
31         << pow( 2.0, 7.0 ) << "\npow(" << 9.0 << ", "
32         << 0.5 << ") = " << pow( 9.0, 0.5 )
33         << setprecision( 3 ) << "\nfmmod("
34         << 13.675 << ", " << 2.333 << ") = "
35         << fmod( 13.675, 2.333 ) << setprecision( 1 )
36         << "\nsin(" << 0.0 << ") = " << sin( 0.0 )

```

```

37         << "\ncos(" << 0.0 << ") = " << cos ( 0.0 )
38         << "\ntan(" << 0.0 << ") = " << tan ( 0.0 ) << endl;
39     return 0;
40 }

```

**输出结果:**

```

Sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675, 2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0

```

- 3.4 a) double hypotenuse( double sidel, double sidel )  
 b) int smallest( int x, int y, int z )  
 c) void instructions( void ) // in C++ (void) can be written()  
 d) float intToFloat( int number )
- 3.5 a) double hypotenuse( double, double )  
 b) int smallest( int, int, int )  
 c) void instructions( void ); // in C++ (void) can be written()  
 d) float intToFloat( int );
- 3.6 a) register int count = 0;  
 b) static float lastVal;  
 c) static int number;  
 注意: 声明在任何函数定义之外。
- 3.7 a) 不正确: 函数 h 在函数 g 中定义。  
 纠正: 将函数 h 移到函数 g 外定义。  
 b) 不正确: 函数要返回整数而没有返回整数。  
 纠正: 删除变量 result 并在函数中加上下列语句:
- ```

return x + y;

```
- c) 不正确: 结果不返回 n + sum(n-1), sum 返回错误结果。

纠正：将 else 中的语句改写为：

```
return n + sum(n - 1);
```

d) 不正确：参数表右括号后面的分号和函数定义中重新定义参数 a。

纠正：删除参数表右括号后面的分号，删除声明语句“float a”；。

e) 不正确：函数不该返回值而返回。值。

纠正：删除 return 语句。

3.8 由于程序员声明 float 引用类型的引用参数，通过引用调用访问原参数变量。

3.9 不正确。C++ 可以用指针和引用参数直接按引用调用。

3.10 如下所示：

```
1 // ex03_10.cpp
2 // Inline function that calculates the volume of a sphere
3 #include <iostream.h>
4
5 const float PI = 3.14159;
6
7 inline float sphere Volume( const float r)
8     { return 4.0 / 3.0 * PI * r * r * r; }
9
10 int main( )
11 {
12     float radius;
13
14     cout << "Enter the length of the radius of your sphere: ";
15     cin >> radius;
16     cout << "Volume of sphere with radius " << radius <<
17         " is " << sphereVolume( radius ) << endl;
18     return 0;
19 }
```

## 练习

3.11 在执行下列每条语句之后显示 x 值：

a) x = fabs( 7.5 )

b) x = floor( 7.5 )

c) x = fabs( 0.0 )

d) x = ceil( 0.0 )

e) x = fabs( -6.4 )

f) x = ceil( -6.4 )

g) x = ceil( -fabs( -8 + floor( -5.5 ) ) )

3.12 在停车场停车 3 个小时以内收费 2 美元，超过三个小时每小时增收 0.5 美元，24 小时最高收费为 10 美元。假设任何车辆一次停车时间不会超过 24 小时。编写一个程序，计算和打印昨天在停车场停车的三个客户的停车费。程序应输入每个客户的停车时间并以整齐的表格形式打印结果，应计算和打印昨天收据的总和。程序用函数 calculateCharges 确定每个客户的停车费。输出格式如下：

| Car   | Hours | Charge |
|-------|-------|--------|
| 1     | 1.5   | 2.00   |
| 2     | 4.0   | 2.50   |
| 3     | 24.0  | 10.00  |
| TOTAL | 29.5  | 14.50  |

3.13 函数 `floor` 的一个应用是将一个值取整为最接近的整数。下列语句：

```
y = floor( x + .5 );
```

将 `x` 值取整为最接近的整数并将结果赋给 `y`。编写一个程序，读取几个数并用上述语句将这些值取整为最接近的整数。对处理的每个数，打印原值和取整值。

3.14 函数 `floor` 可以将一个值取整为特定小数位。下列语句：

```
y = floor( x * 10 + .5 ) / 10;
```

将 `x` 取整为小数点后面第一位（十分位）。下列语句：

```
y = floor( x * 100 + .5 ) / 100;
```

将 `x` 取整为小数点后面第二位（百分位）。编写一个程序，定义 4 个函数，用不同方法取整 `x`：

a) `roundToInteger(number)`

b) `roundToTenths(number)`

c) `roundToHundredths(number)`

d) `roundToThousandths(number)`

对读取的每个值，程序应打印原值、该值取整为最接近的整数、该值取整为最接近的十分位、该值取整为最接近的百分位、该值取整为最接近的千分位。

3.15 回答下列问题：

a) “随机” 选择数值是什么意思？

b) 为什么 `rand` 函数可以模拟 “机会游戏”？

c) 为什么用 `srand` 将程序随机化？在什么情况下最好不要随机化？

d) 为什么通常要位移和缩放 `rand` 产生的值？

e) 为什么用计算机处理模拟现实世界是个有用的技术？

3.16 编写将下列范围的随机整数赋给变量 `n` 的语句。

a)  $1 \leq n \leq 2$

b)  $1 \leq n \leq 100$

c)  $0 \leq n \leq 9$

d)  $1000 \leq n \leq 1112$

e)  $-1 \leq n \leq 1$

f)  $-3 \leq n \leq 11$

3.17 对下列每组整数，编写一个随机打印这些值的语句。

a) 2、4、6、8、10。

b) 3、5、7、9、11。

c) 6、10、14、18、22。

3.18 编写函数 `integerPower(base, exponent)`，返回下列代数式的值：

$\text{base}^{\text{exponent}}$

例如,  $\text{integerPower}(3, 4) = 3 * 3 * 3 * 3$ 。假设  $\text{exponent}$  为正的非 0 整数,  $\text{base}$  为整数。函数  $\text{integerPower}$  用 `for` 或 `while` 控制计算, 不要用任何数学库函数。

- 3.19 定义函数  $\text{hypotenuse}$ , 计算已知两边时直角三角形的弦长, 用这个函数在程序中确定下列三角形的弦长。函数取两个 `double` 类型参数, 返回 `double` 类型值。

| 三角形 | Side1 | Side2 |
|-----|-------|-------|
| 1   | 3.0   | 4.0   |
| 2   | 5.0   | 12.0  |
| 3   | 8.0   | 15.0  |

- 3.20 编写函数  $\text{multiple}$ , 确定一对整数中第二个整数是否为第一个整数的倍数。函数取两个整数参数, 如果第二个整数是第一个的倍数则返回 `true`, 否则返回 `false`。在程序中输入一系列整数, 并使用该函数。
- 3.21 编写一个函数, 输入一组整数, 一次一个地传入函数  $\text{even}$ , 用求模运算符确定其是否为偶数。函数取整数参数, 并在该数为偶数时返回 `true`, 否则返回 `false`。
- 3.22 编写一个函数, 在屏幕左边显示由星形组成的实心正方形, 指定边长为整型参数  $\text{side}$ 。例如,  $\text{side}$  等于 4 时, 函数显示:

```
*****
*****
*****
*****
```

- 3.23 修改练习 3.22 生成的函数, 用  $\text{fillCharacter}$  字符参数中包含的字符填充这个长方形。例如,  $\text{side}$  为 5 而  $\text{fillCharacter}$  为 “#” 时的输出如下:

```
#####
#####
#####
#####
#####
```

- 3.24 用上面两个练习所用的方法产生一个程序, 可以绘制各种图形。

- 3.25 编写完成下列任务的程序段:

- 计算整数  $a$  除以整数  $b$  的商的整数部分。
- 计算整数  $a$  除以整数  $b$  的余数。
- 用 a) 和 b) 中的程序段编写一个程序, 输入 1 到 32767 之间的整数, 打印成一系列数字, 每一对数字之间用两个空格分开。例如, 整数 4562 打印为:

```
4    5    6    2
```

- 3.26 编写一个函数, 将时间设为三个整数参数 (时、分、秒) 并返回从最近的 12 点整算起的秒数。利用这个函数计算两个时间之间的秒数, 时间间隔不超过 12 小时。

- 3.27 实现下列整型函数:

- 函数  $\text{celsius}$  返回华氏温度对应的摄氏温度。
- 函数  $\text{fahrenheit}$  返回摄氏温度对应的华氏温度。
- 用这些函数编写一个程序, 打印从 0 到 100 度的所有摄氏温度及对应的华氏温度, 32 到 212 度的所有华氏温度及对应的摄氏温度。用整齐的表格形式输出, 减少输出行数, 同时保持可读性。

- 3.28 编写一个函数, 返回三个浮点数中的最小值。

- 3.29 一个整数的所有因子（包括1，但不包括本身）之和等于该数，则该数称为完数。例如，6是个完数，因为 $6=1+2+3$ 。编写一个 perfect 程序，确定参数 number 是否为完数。用这个函数确定和打印1到1000之间的所有完数。打印每个完数的因子并确定其的确是个完数。测试比1000大得多的数，挑战计算机的计算能力。
- 3.30 只能被1和自己整除的数称为质数。例如，2、3、5、7就是质数，而4、6、8、9则不是。
- a) 编写一个程序，确定参数是否为质数。
  - b) 用这个函数确定和打印1到10 000之间的所有质数。要先测试多少个数才能确定找出了所有质数？
  - c) 最初你可能以为  $n/2$  是确定找出了所有质数时要测试的数值个数上限，但实际上只要次即可，为什么？改写程序，用两种方法运行，看看性能提高了多少？
- 3.31 编写一个函数，取整数值并返回将数字反序的数值。例如，输入7631，函数返回1367。
- 3.32 两个整数最大公约数（GCD）是这两个数能够整除的最大整数。编写一个 gcd 函数，返回两个整数最大公约数。
- 3.33 编写函数 qualityPoints，输入学生的平均成绩，如果平均成绩在90~100，则返回4，如果平均成绩在80~89，则返回3，如果平均成绩在70~79，则返回2，如果平均成绩在60~69，则返回1，如果平均成绩在60以下，则返回0。
- 3.34 编写一个模拟投币的程序，每次结果应为正面或反面，打印 Heads 或 Tails。让程序投币100次，计算每面出现的次数并打印结果。程序应调用一个 flip 函数，该函数不取参数，返回0表示正面，1表示反面。说明：如果程序真实模拟投币，则每一面出现的次数应近似相等。
- 3.35 计算机在教育领域的作用越来越大，编写一个程序，帮助小学生学习乘法。用 rand 函数产生两个一位正整数，然后输入下列问题：

How much is 6 times 7?

然后学生输入答案，程序检查学生的答案。如果正确，则打印 "Very good!"，然后提出另一个乘法问题。如果不正确，则打印 "No. Please try again."，然后让学生重复回答这个问题，直到答对。

- 3.36 在教学中使用计算机称为计算机辅助教学（CAI）。CAI环境中出现的一个问题是学生容易疲劳。要消除这个问题，可以改变计算机的对话来保持学生的注意力。修改练习3.35的程序，使每次学生答对时和答错时打印不同的评语。

答对时打印：

```
Very good!  
Excellent!  
Nice work!  
Keep up the good work!
```

答错时打印：

```
No. Please try again.  
Wrong. Try once more.  
Don't give up!  
No. Keep trying.
```

用随机数产生器选择1到4的数，由此选择相应评语。用 switch 结构发出响应。

- 3.37 更复杂的计算机辅助教学 (CAI) 系统要监视学生在一段时间的成绩。新内容的推出通常是在学生学好旧内容之后进行的。修改练习3.36的程序, 计算学生答对和答错的比例。学生输入10个答案后, 程序计算其答对率。如果比例不到75%, 则程序打印 "Please ask your instructor for extra help", 然后终止。

- 3.38 编写一个猜数字游戏的程序, 程序随机选择一个1到1000的数, 然后输入:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
```

然后游戏者输入第一个结果。程序响应如下:

```
1. Excellent! You guessed the number!
   Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.
```

如果游戏者猜错, 则程序进行循环, 直到猜对。程序通过 Too high 或 Too low 消息帮助学生接近正确答案。说明: 这个程序中采用的查找技术称为折半查找, 将在下一题更多地讨论。

- 3.39 修改练习3.38的程序, 计算游戏者已猜过的次数。如果次数值为10以下, 则打印 Either you know the secret or you got lucky!。如果10次猜中, 则打印 Ahah! You know the secret!。如果超过10次才猜中, 则打印 You should be able to do better!。为什么不应超过10次呢? 高手怎么在最少的次数内猜中呢? 为什么1到1000的数字能在10次之内猜中呢?

- 3.40 编写一个递归函数 power(base, exponent), 调用时返回

$\text{base}^{\text{exponent}}$

例如,  $\text{power}(3, 4) = 3 * 3 * 3 * 3$ 。假设 exponent 是大于或等于1的整数值。提示: 递归步骤使用下列关系:

$\text{base}^{\text{exponent}} = \text{base} * \text{base}^{\text{exponent}-1}$

并在 exponent 等于1时停止递归, 因为:

$\text{base}^1 = \text{base}$

- 3.41 Fibonacci 数列:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

以0和1开头, 后面的数是前面相邻两个数之和。a)编写一个非递归函数 fibonacci(n), 计算第n个Fibonacci数。b)确定系统中可以打印的最大Fibonacci数。修改程序以使用double类型的整数而不是int类型, 然后重复b)。

- 3.42 (汉诺塔问题) 每个计算机科学家都要掌握一些经典问题, 图3.28所示的汉诺塔问题是最著名的经典问题之一。传说远东地区有一座庙, 僧人要把盘子从一堆移到另一堆。最初一堆在柱子上串了64个盘子, 从下向上逐渐减小。僧人要把这堆移到另一堆, 但一次只能移一个盘子, 且任何时候不能把大盘子放在小盘子上面。第三根柱子可以暂时放置盘子。假设盘子移完之后, 世界末日就会到达, 因此我们不想帮他加快速度。



图 3.28 4 个盘子的汉诺塔问题

假设僧人要把盘子从第 1 根柱子移到第 3 根柱子。我们要开发一个算法，打印移动盘子的顺序。

如果用传统方法解决这个问题，则很快就会发现这个问题太复杂了。如果用递归法，则可以理出思路。移动  $n$  个盘子可以简化为移动  $n-1$  个盘子，如下所示：

- a) 将  $n-1$  个盘子从第 1 根柱子移到第 2 根柱子上，用第 3 根柱子作为临时存放区。
- b) 将最后一个盘子（最大）从第 1 根柱子移到第 3 根柱子。
- c) 将  $n-1$  个盘子从第 2 根柱子移到第 3 根柱子上，用第 1 根柱子作为临时存放区。

移动  $n=1$  个盘子时，过程结束。这时很容易移动盘子，不需要临时存放区。

编写一个解决汉诺塔问题的程序，用递归函数和 4 个函数：

- a) 要移的盘子个数
- b) 最初放盘子的柱子
- c) 盘子移动到该处的柱子
- d) 临时放盘子的柱子

程序应打印从原柱子向目标柱子移动盘子的详细步骤。例如，要将三个盘子从第 1 根柱子移到第 3 根柱子，步骤如下：

1 → 3（即把一个盘子从第 1 根柱子移到第 3 根柱子）

1 → 2

3 → 2

1 → 3

2 → 1

2 → 3

1 → 3

- 3.43 任何能递归实现的程序也可以迭代实现，但有时会比较困难。试编写一个汉诺塔问题的迭代程序。如果成功，试比较迭代程序与练习 3.42 中的递归程序，观察一下程序的性能、清晰性和正确性。

- 3.44（图形化递归）可以观察递归的“操作过程”。修改图 3.14 的阶乘函数，打印其局部函数和递归调用参数。对每个递归调用，单独在一行显示输出并加一级缩排，尽量使输出更



清晰、更有趣、更有意义。这里的目标是设计和实现能让人更容易理解递归的输出格式。可以在许多其他递归练习和例子中增加这种显示功能。

- 3.45 整数  $x$  和  $y$  的最大公约数是  $x$  和  $y$  能够整除的最大整数。编写一个递归函数 `gcd`，返回整数  $x$  和  $y$  的最大公约数。整数  $x$  和  $y$  的最大公约数的递归定义如下：如果  $y$  等于 0，则 `gcd(x, y)` 为  $x$ ，否则 `gcd(x, y)` 为 `gcd(y, x%y)`，其中 `%` 是求模运算符。
- 3.46 `main` 能否递归调用？编写一个包含函数 `main` 的程序。包括 `static` 类型的局部变量 `count` 并初始化为 1。每次调用 `main` 时先递增 `count` 并打印 `count` 的值。运行程序，看看会发生什么情况。
- 3.47 练习 3.35 到 3.37 开发了一个教小学生学乘法的计算机辅助教学程序。本练习要改进这个程序。
- a) 修改程序，让用户输入级别。1 级表示只用一位数，2 级表示可用两位数等等。
  - b) 修改程序，让用户输入要学习的算术类型，1 表示加法、2 表示减法、3 表示乘法、4 表示除法、5 表示混合运算。
- 3.48 编写函数 `distance`，计算两点  $(x_1, y_1)$  与  $(x_2, y_2)$  之间的距离。所有数字和返回值应为 `float` 类型。
- 3.49 下列程序有什么作用？

```
1 // ex3.49.cpp
2 #include <iostream.h>
3
4 int main( )
5 {
6     int c;
7
8     if ( ( c = cin.get( ) ) != EOF ) {
9         main( );
10        cout << c;
11    }
12
13    return 0;
14 }
```

- 3.50 下列程序有什么作用？

```
1 // ex03.50.cpp
2 #include <iostream.h>
3
4 int mystery(int, int );
5
6 int main( )
7 {
8     int x, y;
9
10    cout << "Enter two integers:";
11    cin >> x >> y;
12    cout << "The result is " << mystery ( x,y ) << endl;
13    return 0;
14 }
15
```

```

16 // Parameter b must be a positive
17 // integer to prevent infinite recursion
18 int mystery ( int a, int b )
19 {
20     if ( b == 1 )
21         return a;
22     else
23         return a + mystery(a, b - 1 );
24 }

```

3.51 确定练习3.50的程序结果后修改程序,使第二个参数为非负值的限制删除之后程序还能正确地工作。

3.52 编写一个程序,测试图 3.2 中的各种数学库函数。让程序打印各种参数值和返回值的对照表。

3.53 找出下列程序段中的错误并说明如何纠正。

```

a) float cube( float );    /* function prototype */
    ...
    cube( float number )    /* function definition */
    {
        return number * number * number;
    }
b) register auto int x = 7;
c) int randomNumber = srand( );
d) float y = 123.45678;
    int x;

    x = y;
    cout << static_cast< float> ( x ) << endl;
e) double square( double number )
    {
        double number;
        return number * number;
    }
f) int sum ( int n )
    {
        if ( n == 0 )
            return 0;
        else
            return n + sum( n );
    }

```

3.54 修改图 3.10 的投骰子程序,加进赌注。将程序中运行投骰子游戏的部分打包成函数。将 bankBalance 初始化为 1000 美元,提示游戏者加进赌注 wager。用 while 循环检查 wager 是否小于或等于 bankBalance,如果不是,则提示用户重新输入 wager,直到 wager 有效。输入正确的 wager 之后,运行投骰子游戏。如果游戏者赢,则在 bankBalance 中增加 wager,并打印新的 bankBalance。如果游戏者输,则在 bankBalance 中减去 wager,并打印新的 bankBalance。检查 bankBalance 是否为 0,如果是,则打印消息 "Sorry.You busted!"。游戏进行时,可以打印一些聊天式消息,如 "Oh, you're going for broke, huh?" 或 "Aw cmon, take a chance!" 或 "You're up big. Now's the time to cash in your chips!"。

3.55 编写一个 C++ 程序,用内联函数 circleArea 提示用户输入圆的半径,并计算和打印圆的面积。

- 3.56 编写一个 C++ 程序，用下面指定的两个函数将 `main` 中定义的变量 `count` 乘以三倍。然后比较两个方法。这两个函数如下所示：
- a) 函数 `tripleByReference` 按值调用传递 `count` 的副本，将副本乘以三倍，然后返回新值。
  - b) `tripleByReference` 通过引用参数按引用调用传递 `count`，通过别名（即参数引用）将 `count` 原副本乘以三倍。
- 3.57 一元作用域运算符有什么作用？
- 3.58 编写一个程序，用函数模板 `min` 确定两个参数的最小值。用整数、字符、浮点数对测试这个程序。
- 3.59 编写一个程序，用函数模板 `max` 确定两个参数的最大值。用整数、字符、浮点数对测试这个程序。
- 3.60 确定下列程序段是否有错，对每个错误，说明如何纠正。说明：有的程序段可能没错。
- a) 

```
template < class A >
int sum( int num1, int num2, int num3 )
{
    return num1 + num2 + num3;
}
```
  - b) 

```
void printResults (int x, int y )
{
    cout << "The sum is " << x + y << '\n';
    return x + y;
}
```
  - c) 

```
template < A >
A product ( A num1, A num2, A num3 )
{
    return num1 * num2 * num3;
}
```
  - d) 

```
double cube ( int );
int cube( int );
```

## 第4章 数 组

### 教学目标

- 介绍数组数据结构
- 介绍用数组存放、排序与查找数值清单与表格
- 介绍如何声明数组、初始化数组和引用数组的各个元素
- 将数组传递到函数中
- 介绍基本排序方法
- 声明和操作多下标数组

### 4.1 简介

本章是介绍数据结构的重要课题。数组 (array) 数据结构由相同类型的相关数据项组成。第6章介绍结构 (structure) 和类 (class), 两者都可以带有不同类型的相关数据项。数组和结构是静态项目, 在整个程序执行期间保持相同长度 (当然, 也可以用自动存储类, 在每次进入和离开定义的块时生成和删除)。第15章介绍链表、队列、堆栈、树之类的动态数据结构如何在程序执行期间改变长度。本章介绍的数组是C语言中的指针数组 (第5章将介绍指针)。第8章“运算符重载”和本书末尾的“标准模板库 (STL)”一章将介绍用面向对象编程技术将数组实现为完全成熟的对象, 这些基于对象的数组比第4章介绍的类C语言的指针数组更安全、更灵活。

### 4.2 数组

数组是具有相同名称和相同类型的一组连续内存地址。要引用数组中的特定位置或元素, 就要指定数组中的特定位置或元素的位置号 (position number)。

图4.1显示了整型数组c。这个数组包含12个元素。可以用数组名加上方括号 ([]) 中该元素的位置号引用该元素。数组中的第一个元素称为第0个元素 (zeroth element)。这样, c数组中的第一个元素为c[0], c数组中的第二个元素为c[1], c数组中的第七个元素为c[6], 一般来说, c数组中的第i个元素为c[i-1]。数组名的规则与其他变量名相同。

方括号中的位置号通常称为下标 (subscript), 下标应为整数或整型表达式。如果程序用整型表达式下标, 则要求值这个整型表达式以确定下标, 例如, 假设a等于5, b等于6, 则下列语句:

```
c [ a + b ] += 2
```

将数组元素c[11]加2。注意带下标的数组名是个左值, 可用于赋值语句的左边。

数组名（注意该数组的所有元素同名，均为 *c*）

|              |             |
|--------------|-------------|
| <b>c[0]</b>  | <b>-45</b>  |
| <b>c[1]</b>  | <b>6</b>    |
| <b>c[2]</b>  | <b>0</b>    |
| <b>c[3]</b>  | <b>72</b>   |
| <b>c[4]</b>  | <b>1543</b> |
| <b>c[5]</b>  | <b>-89</b>  |
| <b>c[6]</b>  | <b>0</b>    |
| <b>c[7]</b>  | <b>62</b>   |
| <b>c[8]</b>  | <b>-3</b>   |
| <b>c[9]</b>  | <b>1</b>    |
| <b>c[10]</b> | <b>6453</b> |
| <b>c[11]</b> | <b>78</b>   |

数组中元素的位置号（下标）

图 4.1 12 个元素的整型数组 *c*

图 4.1 中整个数组的名称为 *c*，该数组的 12 个元素为 *c*[0]、*c*[1]、*c*[2]...*c*[11]。*c*[0] 的值为 -45、*c*[1] 的值为 6、*c*[2] 的值为 0，*c*[7] 的值为 62、*c*[11] 的值为 78。要打印数组 *c* 中前三个元素的和，用下列语句：

```
cout << c[ 0 ] + c[ 1 ] + c[ 2 ] << endl;
```

要将数组 *c* 的第 7 个元素的值除以 2，并将结果赋给变量 *x*，用下列语句：

```
x = c[ 6 ] / 2;
```

#### 常见编程错误 4.1

一定要注意“数组的第 7 个元素”与“数组元素 7”之间的差别。由于数组下标从 0 开始，因此“数组第 7 个元素”的下标为 6，而“数组元素 7”的下标为 7，是第 8 个元素。这常常是“差 1 错误”的原因。

包括数组下标的方括号实际上是个 C++ 运算符。方括号的优先级与括号相同。图 4.2 显示了本书前面介绍的 C++ 运算符优先级和结合律。运算符优先级从上到下逐渐减少。

| 运算符                             | 结合律  | 类型      |
|---------------------------------|------|---------|
| () []                           | 从左向右 | 括号      |
| ++ -- + - ! static_cast<type>() | 从右向左 | 一元      |
| * / %                           | 从左向右 | 乘       |
| + -                             | 从左向右 | 加       |
| << >>                           | 从左向右 | 插入 / 读取 |

(续表)

| 运算符              | 结合律  | 类型  |
|------------------|------|-----|
| < <= > >=        | 从左向右 | 关系  |
| == !=            | 从左向右 | 相等  |
| &&               | 从左向右 | 逻辑与 |
|                  | 从左向右 | 逻辑或 |
| ?:               | 从右向左 | 条件  |
| = += -= *= /= %= | 从右向左 | 赋值  |
| ,                | 从左向右 | 逗号  |

图 4.2 运算符的优先级和结合律

### 4.3 声明数组

数组要占用内存空间。程序员指定每个元素的类型和每个数组所要的元素,使编译器可以保留相应的内存空间。要告诉编译器对整型数组 c 保留 12 个元素,可以声明如下:

```
int c[ 12 ];
```

可以在一个声明中为几个数组保留内存。下列声明对整型数组 b 保留 100 个元素,对整型数组 x 保留 27 个元素:

```
int b[ 100 ], x[ 27 ];
```

数组可以声明包含其他数据类型。例如, char 类型的数组可以存放字符串。字符串及其与字符数组的相似性(C++从C语言继承的关系)和指针与数组的关系将在第5章介绍。在介绍面向对象编程后,我们将讨论成熟的字符串对象。

### 4.4 使用数组的举例

图4.3的程序用for重复结构将10个元素的整型数组n的元素初始化为0,并用表格形式打印数组。第一个输出语句显示for结构中所打印列的列标题。记住,setw指定下一个值的输出域宽。

可以在数组声明中用等号和逗号分隔的列表(放在花括号中)将数组中的元素初始化。程序4.4将七个元素的整型数组初始化并用表格形式打印数组。

```
1 // Fig. 4.3: fig04_03.cpp
2 // initializing an array
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int i, n[ 10 ];
9
10    for ( i = 0; i < 10; i++ )           // initialize array
11        n[ i ] = 0;
12
13    cout << "Element" << setw( 13 ) << "Value" << endl;
14
15    for ( i = 0; i < 10; i++ )           // print array
```

```

16     cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
17
18     return 0;
19 }

```

**输出结果:**

| Element | Value |
|---------|-------|
| 0       | 0     |
| 1       | 0     |
| 2       | 0     |
| 3       | 0     |
| 4       | 0     |
| 5       | 0     |
| 6       | 0     |
| 7       | 0     |
| 8       | 0     |
| 9       | 0     |

图 4.3 将 10 个元素的整型数组 n 的元素初始化为 0

```

1 // Fig. 4.4: fig04_04.cpp
2 // Initializing an array with a declaration
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
9
10    cout << "Element" << setw( 13 ) << "Value" << endl;
11
12    for ( int i = 0; i < 10; i++ )
13        cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
14
15    return 0;
16 }

```

**输出结果:**

| Element | Value |
|---------|-------|
| 0       | 32    |
| 1       | 27    |
| 2       | 64    |
| 3       | 18    |
| 4       | 95    |
| 5       | 14    |
| 6       | 90    |
| 7       | 70    |
| 8       | 60    |
| 9       | 37    |

图 4.4 用声明将数组中的元素初始化

如果初始化的元素比数组中的元素少,则其余元素自动初始化为0。例如,可以用下列声明将图 4.3 中数组 n 的元素初始化为 0:

```
int n[ 10 ] = { 0 }
```

其显式地将第一个元素初始化为0，隐式地将其余元素自动初始化为0，因为初始化值比数组中的元素少。程序员至少要显式地将第一个元素初始化为0，才能将其余元素自动初始化为0。图4.3的方法可以在程序执行时重复进行。

#### 常见编程错误 4.2

需要初始化数组元素而没有初始化数组元素是个逻辑错误。

下列数组声明是个逻辑错误：

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

因为有6个初始化值，而数组只有5个元素。

#### 常见编程错误 4.3

初始化值超过数组元素个数是个逻辑错误。

如果带初始化值列表的声明中省略数组长度，则数组中的元素个数就是初始化值列表中的元素个数。例如：

```
int n[] = { 1, 2, 3, 4, 5 };
```

生成五个元素的数组。

#### 性能提示 4.1

如果不用执行时的赋值语句初始化数组而用数组初始化值列表在编译时初始化数组，则程序执行速度更快。

图4.5的程序将10个元素的数组s初始化为整数2、4、6、…20，并以表格形式打印数组。这些数值是将循环计数器的值乘以2再加上2产生的。

```
1 // Fig. 4.5: fig04_05.cpp
2 // Initialize array s to the even integers from 2 to 20.
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     const int arraySize = 10;
9     int j, s[ arraySize ];
10
11     for ( j = 0; j < arraySize; j++ )    // set the values
12         s[ j ] = 2 + 2* j;
13
14     cout << "Element" << setw( 13 ) << "Value" << endl;
15
16     for ( j = 0; j < arraySize; j++ )    // print the values
17         cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
18
19     return 0;
20 }
```

#### 输出结果：

```
Element          Value
```



---

|   |    |
|---|----|
| 0 | 2  |
| 1 | 4  |
| 2 | 6  |
| 3 | 8  |
| 4 | 10 |
| 5 | 12 |
| 6 | 14 |
| 7 | 16 |
| 8 | 18 |
| 9 | 20 |

图 4.5 将产生的值赋给数组元素

下列语句:

```
const int arraySize = 10
```

用 `const` 限定符声明常量变量 `arraySize` 的值为 10。常量变量应在声明时初始化为常量表达式, 此后不能改变(图 4.6 和图 4.7)。常量变量也称为命名常量(named constant)或只读变量(read-only variable)。注意, 常量变量一词是自相矛盾的, 称为逆喻(像“龙虾”之类的名词)。

```
1 // Fig. 4.6: fig04_06.cpp
2 // Using a properly initialized constant variable
3 #include <iostream.h>
4
5 int main()
6 {
7     const int x = 7; // initialized constant variable
8
9     cout << "The value of constant variable x is: "
10         << x << endl;
11
12     return 0;
13 }
```

**输出结果:**

```
The value of constant variable x is: 7
```

图 4.6 正确地初始化和使用常量变量

```
1 // Fig. 4.7: fig04_07.cpp
2 // A const object must be initialized
3
4 int main()
5 {
6     const int x; // Error: x must be initialized
7
8     x = 7; // Error: cannot modify a const variable
9
10    return 0;
11 }
```

**输出结果:**

```
Compiling FIG04-7.CPP:
```

```
Error FIG04_7.CPP 6: Constant variable 'x' must be initialized
Error FIG04_7.CPP 8: Cannot modify a const object
```

图 4.7 const 对象应初始化

**常见编程错误 4.4**

在执行语句中对常量变量赋值是个语法错误。

常量变量可以放在任何出现常量表达式的地方。图 4.5 中，用常量变量 `arraySize` 指定数组 `s` 的长度：

```
int j, s[ arraySize ];
```

**常见编程错误 4.5**

只能用常量声明自动和静态数组，否则是个语法错误。

用常量变量声明数组长度使程序的伸缩性更强。图 4.5 中，要让第一个 `for` 循环填上 1000 个数组元素，只要将 `arraySize` 的值从 10 变为 1000 即可。如果不用常量变量 `arraySize`，则要在程序中进行三处改变才能处理 1000 个数组元素。随着程序加大，这个方法在编写清晰的程序中越来越有用。

**软件工程视点 4.1**

将每个数组的长度定义为常量变量而不是常量，能使程序的伸缩性更强。

**编程技巧 4.1**

将每个数组的长度定义为常量变量而不是常量，能使程序更清晰。这个方法可以取消“魔数”，例如，在处理 10 元素数组的程序中重复出现长度 10 使数字 10 人为地变得重要，程序中存在的与数组长度无关的其他数字 10 时可能使读者搞乱。

图 4.8 中的程序求 12 个元素的整型数组 `a` 中的元素和，`for` 循环体中的语句进行求和。请注意，数组 `a` 的初始化值通常是用户从键盘输入的。例如，下列 `for` 结构：

```
for ( int j = 0; j < arraySize; j++ )
    cin >> a[ j ];
```

一次一个地从键盘读取数值，并将数值存放在元素 `a[j]` 中。

下一个例子用数组汇总调查中收集的数据。考虑下列问题：

40 个学生用 1 到 10 的分数评价学生咖啡屋中的食品质量（1 表示很差，10 表示很好）。将 40 个值放在整型数组中，并汇总调查结果。

这是典型的数组应用（如图 4.9）。我们要汇总每种回答（1 到 10）的个数。数组 `responses` 是 40 个元素的评分数组。我们用 11 个元素的数组 `frequency` 计算每个答案的个数，忽略第一个元素 `frequency[0]`，因为用 1 分对应 `frequency[1]` 而不是对应 `frequency[0]` 更好理解。这样可以直接用回答的分数作为 `frequency` 数组的下标。

```
1 // Fig. 4.8: fig04_08.cppf
2 // Compute the sum of the elements of the array
3 #include <iostream.h>
4
5 int main()
```

```

6 {
7     const int arraySize = 12;
8     int a[ arraySize ] = { 1, 3, 5, 4, 7, 2, 99,
9                           16, 45, 67, 89, 45 };
10    int total = 0;
11
12    for ( int i = 0; i < arraySize ; i++ )
13        total += a[ i ];
14
15    cout << "Total of array element values is " << total << endl;
16    return 0;
17 }

```

**输出结果:**

Total of array element values is 383

图 4.8 计算数组元素和

```

1 // Fig. 4.9: fig04_09.cpp
2 // Student poll program
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     const int responseSize = 40, frequencySize = 11;
9     int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
10    10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
11    5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
12    int frequency[ frequencySize ] = { 0 };
13
14    for ( int answer = 0; answer < responseSize; answer++ )
15        ++frequency[ responses[ answer ] ];
16
17    cout << "Rating" << setw( 17 ) << "Frequency" << endl;
18
19    for ( int rating = 1; rating < frequencySize; rating++ )
20        cout << setw( 6 ) << rating
21            << setw( 17 ) << frequency[ rating ] << endl;
22
23    return 0;
24 }

```

**输出结果:**

| Rating | Frequency |
|--------|-----------|
| 1      | 2         |
| 2      | 2         |
| 3      | 2         |
| 4      | 2         |
| 5      | 5         |
| 6      | 11        |
| 7      | 5         |
| 8      | 7         |
| 9      | 1         |
| 10     | 3         |

图 4.9 学生调查分析程序

#### 编程技巧 4.2

努力保证程序的清晰性,有时甚至可以为保证程序清晰而牺牲内存与处理器时间的使用效率。

#### 性能提示 4.2

有时性能考虑比保证程序的清晰性更重要。

第一个 for 循环一次一个地从 responses 数组取得回答,并将 frequency 数组中的 10 个计数器 (frequency[1]到 frequency[10]) 之一加 1。这个循环中的关键语句如下:

```
++frequency[ responses[ answer ] ];
```

这个语句根据 responses[ answer ] 的值相应递增 frequency 计数器。例如,计数器 answer 为 0 时, responses [ answer ] 为 1, 因此 “++frequency[ responses[ answer ]];” 实际解释如下:

```
++frequency [ 1 ];
```

将数组下标为 1 的元素加 1。计数器 answer 为 1 时, responses[ answer ] 为 2, 因此 “++frequency[ responses [ answer ]];” 实际解释如下:

```
++frequency [ 2 ];
```

将数组下标为 2 的元素加 1。计数器 answer 为 2 时, responses[ answer ] 为 6, 因此 “++frequency[ responses [ answer ]];” 实际解释如下:

```
++frequency [ 6 ];
```

将数组下标为 6 的元素加 1 等等。注意, 不管调查中处理多少个回答, 都只需要 11 个元素的数组 (忽略元素 0) 即可汇总结果。如果数据中包含 13 之类的无效值, 则程序对 frequency [ 13 ] 加 1, 在数组边界之外。C++ 没有数组边界检查, 无法阻止计算机到引用不存在的元素。这样执行程序可能超出数组边界, 而不产生任何警告。程序员应保证所有数组引用都在数组边界之内。C++ 是个可扩展语言, 第 8 章介绍扩展 C++, 用类将数组实现为用户自定义类型。新的数组定义能进行许多 C++ 内建数组中没有的操作, 例如, 可以直接比较数组, 将一个数组赋给另一数组, 用 cin 和 cout 输入和输出整个数组, 自动初始化数组, 防止访问超界数组元素和改变下标范围 (甚至改变下标类型), 使数组第一个元素下标不一定为 0。

#### 常见编程错误 4.6

引用超出数组边界的元素是个执行时的逻辑错误, 而不是语法错误。

#### 测试与调试提示 4.1

对数组进行循环操作时, 数组下标不能低于 0, 不能高于数组中的元素总数 (应比数组中的元素总数少 1)。保证避免循环终止条件访问超界数组元素。

#### 测试与调试提示 4.2

程序应验证所有输入值的正确性, 防止错误信息影响程序计算。

#### 可移植性提示 4.1

访问超界数组元素所带来的影响 (通常很严重) 是与系统有关。

#### 测试与调试提示 4.3

第 6 章开始介绍类时, 将介绍如何开发 “智能” 数组, 可以在运行时自动检查所有下标引用在边界之内。使用这种智能数组能消除一定的缺陷。

下一个例子(图4.10)从数组读取数值,并用条形图或直方图进行描述。打印每个数值,然后在数字旁边打印该数字所指定星号个数的条形图。嵌套for循环实现绘制条形图。注意用endl结束直方图。

#### 常见编程错误4.7

尽管一个for循环和另一个嵌套for循环中可以用相同计数器变量,但这通常是个逻辑错误。

```

1 // Fig. 4.10: fig04_10.cpp
2 // Histogram printing program
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     const int arraySize = 10;
9     int n[ arraySize ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
10
11     cout << "Element" << setw( 13 ) << "Value"
12         << setw( 17 ) << "Histogram" << endl;
13
14     for ( int i = 0; i < arraySize ; i++ ) {
15         cout << setw( 7 ) << i << setw( 13 )
16             << n[ i ] << setw( 9 );
17
18         for ( int j = 0; j < n[ i ]; j++ )    // print one bar
19             cout << '*';
20
21         cout << endl;
22     }
23
24     return 0;
25 }

```

#### 输出结果:

| Element | Value | Histogram |
|---------|-------|-----------|
| 0       | 19    | *****     |
| 1       | 3     | ***       |
| 2       | 15    | *****     |
| 3       | 7     | *****     |
| 4       | 11    | *****     |
| 5       | 9     | *****     |
| 6       | 13    | *****     |
| 7       | 5     | *****     |
| 8       | 17    | *****     |
| 9       | 1     | *         |

图4.10 打印直方图的程序

#### 测试与调试提示4.4

尽管for循环中的计数器变量可以修改,但这样容易造成一些缺陷,应当避免。

第3章介绍了投骰子程序(参见图3.8),就是将六面体骰子投6000次,测试随机数产生器是否真的产生随机数。图4.11显示了这个程序的数组版本。

```

1 // Fig. 4.11: fig04_11.cpp
2 // Roll a six-sided die 6000 times
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6 #include <time.h>
7
8 int main()
9 {
10     const int arraySize = 7;
11     int face, frequency[ arraySize ] = { 0 };
12
13     srand( time( 0 ) );
14
15     for ( int roll = 1; roll <= 6000; roll++ )
16         ++frequency[ 1 + rand() % 6 ]; // replaces 20-line switch
17   // of Fig. 3.8
18
19     cout << "Face" << setw( 13 ) << "Frequency" << endl;
20
21     // ignore element 0 in the frequency array
22     for ( face = 1; face < arraySize ; face++ )
23         cout << setw( 4 ) << face
24             << setw( 13 ) << frequency[ face ] << endl;
25
26     return 0;
27 }

```

**输出结果：**

| Face | Frequency |
|------|-----------|
| 1    | 1037      |
| 2    | 987       |
| 3    | 1013      |
| 4    | 1028      |
| 5    | 952       |
| 6    | 983       |

图 4.11 用数组而不用 switch 结构设计投骰子程序

前面只介绍了整型数组，但数组可以是任何类型，下面要介绍如何用字符数组存放字符串。前面介绍的字符串处理功能只有用 cout 和 << 输入字符串，“hello”之类的字符串其实就是一个字符数组。字符数组有几个特性。

字符数组可以用字符串直接量初始化。例如，下列声明：

```
char string1[] = "first";
```

将数组 string1 的元素初始化为字符串“first”中的各个元素。上述声明中 string1 的长度是编译器根据字符串长度确定的。注意，字符串“first”包括五个字符加一个特殊字符串终止符，称为空字符（null character），这样，字符串 string1 实际上包含 6 个元素。空字符对应的字符常量为 '\0'（反斜杠加 0），所有字符串均用这个空字符结尾。表示字符串的字符数组应该足以放置字符串中的所有字符和空字符。

字符数组还可以用初始化值列表中的各个字符常量初始化。上述语句也可以写成：

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

由于字符串其实是字符数组,因此可以用数组下标符号直接访问字符串中的各个字符。例如, `string1[0]` 是字符 'f', `string1[3]` 是字符 's'。

我们还可以用 `cin` 和 `>>` 直接从键盘输入字符数组。例如, 下列声明:

```
char string2[ 20 ];
```

生成的字符数组能存放 19 个字符和一个 `null` 终止符的字符串。

下列语句:

```
cin >> string2;
```

从键盘中将字符串读取到 `string2` 中。注意, 上述语句中只提供了数组名, 没有提供数组长度的信息。程序员要负责保证接收字符串的数组能够放置用户从键盘输入的任何字符串。`cin` 从键盘读取字符, 直到遇到第一个空白字符, 它不管数组长度如何。这样, 用 `cin` 和 `>>` 输入数据可能插入到数组边界之外 (5.12 节介绍如何防止插入到数组边界之外)。

#### 常见编程错误 4.8

如果 `cin>>` 不提供足够大的数组, 则键盘输入时可能造成数据丢失和其他严重的运行时错误。

可以用 `cout` 和 `<<` 输出表示空字符终止字符串的字符数组。下列语句打印数组 `string2`:

```
cout << string2 << endl;
```

注意 `cout<<` 和 `cin>>` 一样不在乎字符数组的长度。一直打印字符串的字符, 直到遇到 `null` 终止符为止。

图 4.12 演示直接用字符串初始化字符数组、将字符串读取到字符数组中、将字符数组作为字符串打印以及访问字符串的各个字符。

```
1 // Fig. 4_12: fig04_12.cpp
2 // Treating character arrays as strings
3 #include <iostream.h>
4
5 int main()
6 {
7     char string1[ 20 ], string2[] = "string literal";
8
9     cout << "Enter a string: ";
10    cin >> string1;
11    cout << "string1 is: " << string1
12         << "\nstring2 is: " << string2
13         << "string1 with spaces between characters is:\n";
14
15    for ( int i = 0; string1[ i ] != '\0'; i++ )
16        cout << string1[ i ] << ' ';
17
18    cin >> string1; // reads "there"
19    cout << "\nstring1 is: " << string1 << endl;
20
21    cout << endl;
22    return 0;
23 }
```

**输出结果:**

```

Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
string1 is: there

```

图 4.12 将字符数组当作字符串

图 4.12 用 for 结构在 string1 数组中循环, 并打印各个字符, 用空格分开。for 结构的条件 string1[i] != '\0' 在遇到字符串的 null 终止符之前一直为真。

第 3 章介绍了存储类说明符 static。函数体中的 static 局部变量在程序执行期间存在, 但只能在函数体中访问该变量。

**性能提示 4.3**

可以用 static 对局部数组声明, 使该数组不必在每次调用函数时生成和初始化, 该数组在程序中每次退出函数时不会删除, 这样可以提高性能。

声明为 static 的数组在程序装入时初始化。如果程序员不把 static 数组显式初始化, 则编译器将这个数组初始化为 0。

图 4.13 显示了带有声明为 static 的局部数组的 staticArrayInit 函数和带自动局部数组的 automaticArrayInit 函数。staticArrayInit 调用两次, 编译器将 static 局部数组初始化为 0。该函数打印数组, 将每个元素加 5, 然后再次打印该数组; 函数第二次调用时, static 数组包含第一次调用时所存放的值。函数 automaticArrayInit 也调用两次, 但自动局部数组的元素用数值 1、2、3 初始化。该函数打印数组, 将每个元素加 5, 然后再次打印该数组; 函数第二次调用时, 数组重新初始化为 1、2、3, 因为这个数组是自动存储类。

```

1 // Fig. 4.13: fig04_13.cpp
2 // Static arrays are initialized to zero
3 #include <iostream.h>
4
5 void staticArrayInit( void );
6 void automaticArrayInit( void );
7
8 int main()
9 {
10     cout << "First call to each function:\n";
11     staticArrayInit();
12     automaticArrayInit();
13
14     cout << "\n\nSecond call to each function:\n";
15     staticArrayInit();
16     automaticArrayInit();
17     cout << endl;
18
19     return 0;
20 }
21
22 // function to demonstrate a static local array
23 void staticArrayInit( void )

```



```

24 {
25     static int array1[ 3 ];
26     int i;
27
28     cout << "\nValues on entering staticArrayInit:\n";
29
30     for ( i = 0; i < 3; i++ )
31         cout << "array1[ " << i << " ] = " << array1[ i ] << " ";
32
33     cout << "\nValues on exiting staticArrayInit:\n";
34
35     for ( i = 0; i < 3; i++ )
36         cout << "array1[ " << i << " ] = "
37             << ( array1[ i ] += 5 ) << " ";
38 }
39
40 // function to demonstrate an automatic local array
41 void automaticArrayInit( void )
42 {
43     int i, array2[ 3 ] = { 1, 2, 3 };
44
45     cout << "\n\nValues on entering automaticArrayInit:\n";
46
47     for ( i = 0; i < 3; i++ )
48         cout << "array2[ " << i << " ] = " << array2[ i ] << " ";
49
50     cout << "\nValues on exiting automaticArrayInit:\n";
51
52     for ( i = 0; i < 3; i++ )
53         cout << "array2[ " << i << " ] = "
54             << ( array2[ i ] += 5 ) << " ";
55 }

```

**输出结果:**

First call to each function:

```

Values on entering staticArrayInit:
array1[ 0 ] = 0 array1[ 1 ] = 0 array1[ 2 ] = 0
Values on exiting staticArrayInit:
array1[ 0 ] = 5 array1[ 1 ] = 5 array1[ 2 ] = 5

```

```

Values on entering automaticArrayInit:
array2[ 0 ] = 1 array2[ 1 ] = 2 array2[ 2 ] = 3
Values on exiting automaticArrayInit:
array2[ 0 ] = 6 array2[ 1 ] = 7 array2[ 2 ] = 8

```

second call to each function:

```

Values on entering staticArrayInit:
array1[ 0 ] = 5 array1[ 1 ] = 5 array1[ 2 ] = 5
Values on exiting staticArrayInit:
array1[ 0 ] = 10 array1[ 1 ] = 10 array1[ 2 ] = 10

```

```

Values on entering automaticArrayInit:
array2[ 0 ] = 1 array2[ 1 ] = 2 array2[ 2 ] = 3

```

```
Values on exiting automaticArrayInit:  
array2[ 0 ] = 6 array2[ 1 ] = 7 array2[ 2 ] = 8
```

图 4.13 比较 static 数组初始化和自动数组初始化

#### 常见编程错误 4.9

假设每次调用函数时函数局部 static 数组的元素初始化为 0 将导致程序中的逻辑错误。

## 4.5 将数组传递给函数

要将数组参数传递给函数,需指定不带方括号的数组名。例如,如果数组 `hourlyTemperatures` 声明如下:

```
int hourlyTemperatures [ 24 ];
```

则下列函数调用语句:

```
modifyArray( hourlyTemperatures, 24);
```

将数组 `hourlyTemperatures` 及其长度传递给函数 `modifyArray`。将数组传递给函数时,通常也将其长度传递给函数,使函数能处理数组中特定的元素个数(否则要在被调用函数中建立这些信息,甚至要把数组长度放在全局变量中)。第 8 章介绍 `Array` 类时,将把数组长度设计在用户自定义类型中,每个 `Array` 对象生成时都“知道”自己的长度。这样,将 `Array` 对象传递给函数时,就不用把数组长度作为参数一起传递。

C++ 使用模拟的按引用调用,自动将数组传递给函数,被调用函数可以修改调用者原数组中的元素值。数组名的值为数组中第一个元素的地址。由于传递数组的开始地址,因此被调用函数知道数组的准确存放位置。因此,被调用函数在函数体中修改数组元素时,实际上是修改原内存地址中的数组元素。

#### 性能提示 4.4

模拟按引用调用传递数组时才能有性能上的意义。如果数组按值传递,则是传递每个元素的副本。对于经常传递的大数组,这是很费时间的,而且存放数组副本要占用很大空间。

#### 软件工程视点 4.2

也可以按值传递数组(用第 6 章介绍的简单方法),但很少使用。

尽管模拟按引用调用传递整个数组,但各个数组元素和简单变量一样是按值传递。这种简单的单个数据称为标量(`scalar` 或 `scalar quantity`)。要将数组元素传递给函数,用数组元素的下标名作为函数调用中的参数。第 5 章将介绍标量(即各个变量和数组元素)的模拟按引用调用。

要让函数通过函数调用接收数组,函数的参数表应指定接收数组。例如,函数 `modifyArray` 的函数首部可能如下所示:

```
void modifyArray( int b[], int arraySize )
```

表示 `modifyArray` 要在参数 `b` 中接收整型数组并在参数 `arraySize` 中接收数组元素个数。数组方括号中的数组长度不是必需的,如果包括,则编译器将其忽略。由于模拟按引用调用传递数组,因此被调用函数使用数组名 `b` 时,实际上引用调用者的实际数组(上例中为数组 `hourlyTemperatures`)。第 5 章介绍表示函数接收数组的其他符号,这些符号基于数组与指针之间的密切关系。

注意 modifyArray 函数原型的表示方法:

```
void modifyArray( int[], int );
```

这个原型也可以改写成:

```
void modifyArray( int anyArrayName[], int anyVariableName )
```

但第3章曾介绍过, C++ 编译器忽略函数原型中的变量名。

#### 编程技巧 4.3

有些程序员在函数原型中包括变量名, 使程序更清晰, 编译器将忽略这个名称。

记住, 函数原型告诉编译器参数个数和参数类型 (按参数出现的顺序)。

图4.14的程序演示了传递整个数组与传递数组元素之间的差别。程序首先打印整型数组a的五个元素, 然后将a及其长度传递给函数 modifyArray, 其中将a数组中的元素乘以2, 然后在main中重新打印a。从输出可以看出, 实际由 modifyArray 修改a的元素。现在程序打印a[3]的值并将其传递给函数 modifyElement。函数 modifyElement 将参数乘以2。然后打印新值。注意在main中重新打印a[3]时, 它没有修改, 因为各个数组元素是按值调用传递。

```
1 // Fig. 4.14: fig04_14.cpp
2 // Passing arrays and individual array elements to functions
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 void modifyArray( int [], int ); // appears strange
7 void modifyElement( int );
8
9 int main()
10 {
11     const int arraySize = 5;
12     int i, a[ arraySize ] = { 0, 1, 2, 3, 4 };
13
14     cout << "Effects of passing entire array call-by-reference:"
15         << "\n\nThe values of the original array are:\n";
16
17     for ( i = 0; i < arraySize; i++ )
18         cout << setw( 3 ) << a[ i ];
19
20     cout << endl;
21
22     // array a passed call-by-reference
23     modifyArray( a, arraySize );
24
25     cout << "The values of the modified array are:\n";
26
27     for ( i = 0; i < arraySize; i++ )
28         cout << setw( 3 ) << a[ i ];
29
30     cout << "\n\n\n"
31         << "Effects of passing array element call-by-value:"
32         << "\n\nThe value of a[ 3 ] is " << a[ 3 ] << '\n';
33 }
```

```

34     modifyElement( a[ 3 ] );
35
36     cout << "The value of a[ 3 ] is " << a[ 3 ] << endl;
37
38     return 0;
39 }
40
41 void modifyArray( int b[], int sizeOfArray )
42 {
43     for ( int j = 0; j < sizeOfArray; j++ )
44         b[ j ] *= 2;
45 }
46
47 void modifyElement( int e )
48 {
49     cout << "Value in modifyElement is "
50         << ( e * 2 ) << endl;
51 }

```

**输出结果:**

Effects of passing entire array call-by-Value:

The values of the original array are:

0 1 2 3 4

The values of the modified array are:

0 2 4 6 8

Effects of passing array element call-by-value:

The value of a[ 3 ] is 6

Value in modifyElement is 12

The value of a[ 3 ] is 6

图 4.14 向函数传递数组和数组元素

有时程序中的函数不能修改数组元素。由于总是模拟按引用调用传递数组,因此数组中数值的修改很难控制。C++ 提供类型限定符 `const`, 可以防止修改函数中的数组值。数组参数前面加上 `const` 限定符时, 数组元素成为函数体中的常量, 要在函数体中修改数组元素会造成语法错误。这样, 程序员就可以纠正程序, 使其不修改数组元素。

图 4.15 演示了 `const` 限定符。函数 `tryToModifyArray` 定义参数 `const int b[]`, 指定数组 `b` 为常量, 不能修改。函数想修改数组元素会造成语法错误 “Cannot modify a const object”。`const` 限定符将在第 7 章再次介绍。

```

1 // Fig. 4.15: fig04_15.cpp
2 // Demonstrating the const type qualifier
3 #include <iostream.h>
4
5 void tryToModifyArray( const int[] );
6
7 int main()
8 {
9     int a[] = { 10, 20, 30 };
10

```

```

11     tryToModifyArray( a );
12     cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
13     return 0;
14 }
15
16 void tryToModifyArray( const int b[] )
17 {
18     b[ 0 ] /= 2;    // error
19     b[ 1 ] /= 2;    // error
20     b[ 2 ] /= 2;    // error
21 }

```

**输出结果：**

```

Compiling FIG04_15.CPP:
Error FIG04_15.CPP 18: Cannot modify a const object
Error FIG04_15.CPP 19: Cannot modify a const object
Error FIG04_15.CPP 20: Cannot modify a const object
Warning FIG04_15.CPP 21: Parameter 'b' is never used

```

图4.15 演示 const 限定符

**常见编程错误 4.10**

忘记数组按引用传递以至于修改数组可能造成逻辑错误。

**软件工程视点 4.3**

const 限定符可以用于函数定义中的数组参数，防止函数体中修改原数组，这是最低权限原则的另一个例子。函数不能提供修改数组的功能，除非确实有必要。

## 4.6 排序数组

排序 (sort) 数组 (即将数据排成特定顺序，如升序或降序) 是一个重要的计算应用。银行按账号排序所有支票，使每个月末可以准备各个银行报表。电话公司按姓氏排序账号清单并在同一姓氏中按名字排序，以便于找到电话号码。几乎每个公司都要排序一些数据，有时要排序大量数据。排序数据是个复杂问题，是计算机科学中大量研究的课题。本章介绍最简单的排序机制，在本章练习和第 15 章中，我们要介绍更复杂的机制以达到更高的性能。

**性能提示 4.5**

有时最简单的算法性能太差，但易于编写、测试和调试。更复杂的算法在需要实现最高性能时才会用到。

图4.16的程序排列 10 个元素数组 a 的值，按升序排列。我们使用冒泡排序 (bubble sort 或 sinking sort) 方法，较少的数值慢慢从下往上“冒”，就像水中的气泡一样，而较大的值则慢慢往下沉。这个方法在数组中多次操作，每一次都比较一对相邻元素。如果某一对为升序 (或数值相等)，则将数值保持不变。如果某一对为降序，则将数值交换。

```

1 // Fig. 4.16: fig04_16.cpp
2 // This program sorts an array's values into
3 // ascending order
4 #include <iostream.h>
5 #include <iomanip.h>

```

```

6
7 int main()
8 {
9     const int arraySize = 10;
10    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
11    int i, hold;
12
13    cout << "Data items in original order\n";
14
15    for ( i = 0; i < arraySize; i++ )
16        cout << setw( 4 ) << a[ i ];
17
18    for ( int pass = 0; pass < arraySize - 1; pass++ ) // passes
19
20        for ( i = 0; i < arraySize - 1; i++ )          // one pass
21
22            if ( a[ i ] > a[ i + 1 ] ) {                // one comparison
23                hold = a[ i ];                          // one swap
24                a[ i ] = a[ i + 1 ];
25                a[ i + 1 ] = hold;
26            }
27
28    cout << "\nData items in ascending order\n";
29
30    for ( i = 0; i < arraySize; i++ )
31        cout << setw( 4 ) << a[ i ];
32
33    cout << endl;
34    return 0;
35 }

```

**输出结果：**

```

Data items in original order
 2   6   4   8  10  12  89  68  45  37
Data items in ascending order
 2   4   6   8  10  12  37  45  68  89

```

#### 4.16 用冒泡法排序数组

程序首先比较  $a[0]$  与  $a[1]$ ，然后比较  $a[1]$  与  $a[2]$ ，接着是  $a[2]$  与  $a[3]$ ，一直到比较  $a[8]$  与  $a[9]$ 。尽管有 10 个元素，但只进行 9 次比较。由于连续进行比较，因此一次即可能将大值向下移动多位，但小值只能向上移动一位。第 1 遍，即可把最大的值移到数组底部，变为  $a[9]$ 。第 2 遍，即可将第 2 大的值移到  $a[8]$ 。第 9 遍，将第 9 大的值移到  $a[1]$ ，最小值即为  $a[0]$ ，因此，只进行 9 次比较即可排序 10 个元素的数组。

排序是用嵌套 for 循环完成的。如果需要交换，则用三条赋值语句完成：

```

hold = a[ i ];
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = hold;

```

其中附加的变量 hold 临时保存要交换的两个值之一。只有两个赋值语句是无法进行交换的：

```

a[ i ] = a[ i + 1 ];

```

```
a[ i + 1 ] = a[ i ];
```

例如, 如果  $a[i]$  为 7 而  $a[i+1]$  为 5, 则第一条赋值语句之后, 两个值均为 5, 数值 7 丢失, 因此要先用变量 `hold` 临时保存要交换的两个值之一。

冒泡排序的主要优点是易于编程。但冒泡排序的速度很慢, 这在排序大数组时更明显。练习中要开发更有效的冒泡排序程序, 介绍一些比冒泡排序更有效的方法。高级课题中将介绍更深入的排序与查找问题。

## 4.7 实例研究：用数组计算平均值、中数和模

下面要举一个更大的例子。计算机常用于编译和分析调查结果, 图 4.17 的程序用数组 `response` 初始化调查的 99 个答复 (用常量变量 `responseSize` 表示), 每个答复是 1 到 9 的数值。程序计算 99 个值的平均值、中数和模。

```
1 // Fig. 4.17: fig04_17.cpp
2 // This program introduces the topic of survey data analysis.
3 // It computes the mean, median, and mode of the data.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 void mean( const int [], int );
8 void median( int [], int );
9 void mode( int [], int [], int );
10 void bubbleSort( int[], int );
11 void printArray( const int[], int );
12
13 int main()
14 {
15     const int responseSize = 99;
16     int frequency[ 10 ] = { 0 },
17         response[ responseSize ] =
18         { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
19           7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
20           6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
21           7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
22           6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
23           7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
24           5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
25           7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
26           7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
27           4, 5, 6, 1, 6, 5, 7, 8, 7 };
28
29     mean( response, responseSize );
30     median( response, responseSize );
31     mode( frequency, response, responseSize );
32
33     return 0;
34 }
35
36 void mean( const int answer[], int arraySize )
37 {
38     int total = 0;
```

```

39
40     cout << "*****\n Mean\n*****\n";
41
42     for ( int j = 0; j < arraySize; j++ )
43         total += answer[ j ];
44
45     cout << "The mean is the average value of the data\n"
46         << "items. The mean is equal to the total of\n"
47         << "all the data items divided by the number\n"
48         << "of data items (" << arraySize
49         << "). The mean value for\nthis run is: "
50         << total << " / " << arraySize << " = "
51         << setiosflags( ios::fixed | ios::showpoint )
52         << setprecision( 4 ) << ( float ) total / arraySize
53         << "\n\n";
54 }
55
56 void median( int answer[], int size )
57 {
58     cout << "\n*****\n Median\n*****\n"
59         << "The unsorted array of responses is";
60
61     printArray( answer, size );
62     bubbleSort( answer, size );
63     cout << "\n\nThe sorted array is";
64     printArray( answer, size );
65     cout << "\n\nThe median is element " << size / 2
66         << " of\nthe sorted " << size
67         << " element array.\nFor this run the median is "
68         << answer[ size / 2 ] << "\n\n";
69 }
70
71 void mode( int freq[], int answer[], int size )
72 {
73     int rating, largest = 0, modeValue = 0;
74
75     cout << "\n*****\n Mode\n*****\n";
76
77     for ( rating = 1; rating <= 9; rating++ )
78         freq[ rating ] = 0;
79
80     for ( int j = 0; j < size; j++ )
81         ++freq[ answer[ j ] ];
82
83     cout << "Response"<< setw( 11 ) << "Frequency"
84         << setw( 19 ) << "Histogram\n\n" << setw( 55 )
85         << "1    1    2    2\n" << setw( 56 )
86         << "5    0    5    0    5\n\n";
87
88     for ( rating = 1; rating <= 9; rating++ ) {
89         cout << setw( 8 ) << rating << setw( 11 )
90             << freq[ rating ] << "    ";
91
92         if ( freq[ rating ] > largest ) {
93             largest = freq[ rating ];

```



```

94         modeValue = rating;
95     }
96
97     for ( int h = 1; h <= freq[ rating ]; h++ )
98         cout << '*';
99
100        cout << '\n';
101    }
102
103    cout << "The mode is the most frequent value.\n"
104          << "For this run the mode is " << modeValue
105          << " which occurred " << largest << " times." << endl;
106 }
107
108 void bubbleSort( int a[], int size )
109 {
110     int hold;
111
112     for ( int pass = 1; pass < size; pass++ )
113
114         for ( int j = 0; j < size - 1; j++ )
115
116             if ( a[ j ] > a[ j + 1 ] ) {
117                 hold = a[ j ];
118                 a[ j ] = a[ j + 1 ];
119                 a[ j + 1 ] = hold;
120             }
121 }
122
123 void printArray( const int a[], int size )
124 {
125     for ( int j = 0; j < size; j++ ) {
126
127         if ( j % 20 == 0 )
128             cout << endl;
129
130         cout << setw( 2 ) << a[ j ];
131     }
132 }

```

图 4.17 调查数据分析程序

平均值是 99 个数的算术平均值。函数 `mean` 计算 99 个元素的和除以 99 所得的平均值。

中数是“中间值”，函数 `median` 调用 `bubbleSort` 函数排序（按升序）答复数组，并选出数组的中间元素 `answer[ responseSize / 2 ]`。注意，如果元素个数为偶数，则中数为中间两个元素的平均值，函数 `median` 目前没有提供这个功能。调用函数 `printArray` 输出 `response` 数组。

模是 99 个答复中最经常出现值。函数 `mode` 计算每种答复的个数，然后选择个数最多的值。这个版本的函数 `mode` 不处理连接（见练习 4.14）。函数 `mode` 还产生直方图，帮助用图形确定模。图 4.18 包含这个程序的示例输出。这个例子包括数组问题中通常需要的最常見操作，包括将数组传递给函数。

```

*****
Mena
*****
The mean is the average value of the data
items.The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788

*****
Mena
*****
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8
 9
The median is element 49 of
The sorted 99 element array.
For this run the median is 7
*****
Mode
*****
Response Frequency      Histogram

                                1  1  2  2
                                5  0  5  0  5

1          1          *
2          3          **
3          4          ***
4          5          ****
5          8          *****
6          9          *
7          23         *****
8          27         *****
9          19         *****
The mode is the most frequent value
For this run the is 8 which occurred 27 times.

```

图 4.18 调查数据分析程序的运行结果

## 4.8 查找数组：线性查找与折半查找

程序员经常要处理数组中存放的大量数据，可能需要确定数组是否包含符合某关键值（key value）的值。寻找数组中某个元素的过程称为查找（searching）。本节介绍两个查找方法：简单的

线性查找 (linear search) 方法和更复杂的折半查找 (binary search) 方法。练习 4.33 和练习 4.34 要求用递归法实现线性查找与折半查找。

图 4.19 的线性查找比较数组中每个元素与查找键 (search key)。由于数组没有特定的顺序, 很可能第一个就找到, 也可能要到最后一个才找到。因此, 平均起来, 程序要比较数组中一半的元素才能找到查找键值。要确定哪个值不在数组中, 则程序要比较查找键与数组中每一个元素。

```
1 // Fig. 4.19: fig04_19.cpp
2 // Linear search of an array
3 #include <iostream.h>
4
5 int linearSearch( const int [], int, int );
6
7 int main()
8 {
9     const int arraySize = 100;
10    int a[ arraySize ], searchKey, element;
11
12    for ( int x = 0; x < arraySize; x++ ) // create some data
13        a[ x ] = 2 * x;
14
15    cout << "Enter integer search key:" << endl;
16    cin >> searchKey;
17    element = linearSearch( a, searchKey, arraySize );
18
19    if ( element != -1 )
20        cout << "Found value in element " << element << endl;
21    else
22        cout << "Value not found" << endl;
23
24    return 0;
25 }
26
27 int linearSearch( const int array[], int key, int sizeOfArray )
28 {
29     for ( int n = 0; n < sizeOfArray; n++ )
30         if ( array[ n ] == key )
31             return n;
32
33     return -1;
34 }
```

**输出结果:**

```
Enter integer search key:
36
found value in element 18
```

```
Enter integer search key:
37
Value not found
```

图 4.19 数组的线性查找

线性查找方法适用于小数组或未排序数组。但是, 对于大数组, 线性查找是低效的。如果是排序数组, 则可以用高速折半查找。

折半查找算法在每次比较之后排除所查找数组的一半元素。这个算法找到数组的中间位置，将其与查找键比较。如果相等，则已找到查找键，返回该元素的数组下标。否则将问题简化为查找一半数组。如果查找键小于数组中间元素，则查找数组的前半部分，否则查找数组的后半部分。如果查找键不是指定子数组（原始数组的一部分）中的中间元素，则对原数组的四分之一重复这个算法。查找一直继续，直到查找键等于指定子数组中的中间元素或子数组只剩一个元素且不等于查找键（表示找不到这个查找键）为止。

在最糟糕的情况下，查找 1024 个元素的数组只要用折半查找进行十次比较。重复将 1024 除 2（因为折半查找算法在每次比较之后排除所查找数组的一半元素）得到值 512、256、128、64、32、16、8、4、2 和 1。数值 1024 ( $2^{10}$ ) 除 2 十次之后即变成 1。除以 2 就是折半查找算法的一次比较。1048576 ( $2^{20}$ ) 个元素的数组最多只要 20 次比较就可以取得查找键。十亿个元素的数组最多只要 30 次比较就可以取得查找键。这比线性查找的性能大有提高，后者平均要求比较数组元素个数一半的次数。对于十亿个元素的数组，这是 5 亿次比较与 30 次比较的差别。折半查找所需的最大比较次数可以通过大于数组元素个数的第一个 2 的次幂的指数确定。

#### 性能提示 4.6

折半查找所带来的巨大的性能提高是有代价的，比起一次查找一个项目，排序数组的成本要高得多。如果数组需要多次高速查找，则排序数组的开销是值得的。

图 4.20 显示了函数 `binarySearch` 的迭代版本。函数取 4 个参数，一个整数数组 `b`、一个整数 `searchKey`、`low` 数组下标和 `high` 数组下标。如果查找键不符合子数组的中间元素，则调整 `low` 数组下标和 `high` 数组下标，以便查找更小的子数组。如果查找键小于中间元素，则 `high` 数组的下标设置为 `middle-1`，继续查找 `low` 到 `middle-1` 的元素。如果查找键大于中间元素，则 `low` 数组的下标设置为 `middle+1`，继续查找 `middle+1` 到 `high` 的元素。程序使用 15 个元素的数组。大于数组元素个数的第一个 2 的次幂是 16 ( $2^4$ )，因此寻找查找键最多只要四次比较。函数 `printHeader` 输出数组下标，函数 `printRow` 输出折半查找过程中的每个子数组。每个子数组的中间元素标上星号 (\*)，表示用这个元素与查找键比较。

```
1 // Fig. 4.20: fig04_20.cpp
2 // Binary search of an array
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int binarySearch( int [], int, int, int, int );
7 void printHeader( int );
8 void printRow( int [], int, int, int, int );
9
10 int main()
11 {
12     const int arraySize = 15;
13     int a[ arraySize ], key, result;
14
15     for ( int i = 0; i < arraySize; i++ )
16         a[ i ] = 2 * i;    // place some data in array
17
18     cout << "Enter a number between 0 and 28: ";
19     cin >> key;
20
21     printHeader( arraySize );
```

```

22     result = binarySearch( a, key, 0, arraySize - 1, arraySize );
23
24     if ( result != -1 )
25         cout << '\n' << key << " found in array element "
26             << result << endl;
27     else
28         cout << '\n' << key << " not found" << endl;
29
30     return 0;
31 }
32
33 // Binary search
34 int binarySearch( int b[], int searchKey, int low, int high,
35                  int size )
36 {
37     int middle;
38
39     while ( low <= high ) {
40         middle = ( low + high ) / 2;
41
42         printRow( b, low, middle, high, size );
43
44         if ( searchKey == b[ middle ] ) // match
45             return middle;
46         else if ( searchKey < b[ middle ] )
47             high = middle - 1;          // search low end of array
48         else
49             low = middle + 1;           // search high end of array
50     }
51
52     return -1;    // searchKey not found
53 }
54
55 // Print a header for the output
56 void printHeader( int size )
57 {
58     cout << "\nSubscripts:\n";
59     for ( int i = 0; i < size; i++ )
60         cout << setw( 3 ) << i << ' ';
61
62     cout << '\n';
63
64     for ( i = 1; i <= 4 * size; i++ )
65         cout << '-';
66
67     cout << endl;
68 }
69
70 // Print one row of output showing the current
71 // part of the array being processed.
72 void printRow( int b[], int low, int mid, int high, int size )
73 {
74     for ( int i = 0; i < size; i++ )
75         if ( i < low || i > high )
76             cout << "    ";

```

```

77         else if ( i == mid )           // mark middle value
78             cout << setw( 3 ) << b[ i ] << '*';
79         else
80             cout << setw( 3 ) << b[ i ] << ' ';
81
82     cout << endl;
83 }

```

#### 输出结果:

Enter a number between 0 and 28: 25

Subscripts:

| 0 | 1 | 2 | 3 | 4 | 5  | 6  | 7   | 8  | 9  | 10 | 11  | 12 | 13  | 14 |
|---|---|---|---|---|----|----|-----|----|----|----|-----|----|-----|----|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14* | 16 | 18 | 20 | 22  | 24 | 26  | 28 |
|   |   |   |   |   |    |    |     | 16 | 18 | 20 | 22* | 24 | 26  | 28 |
|   |   |   |   |   |    |    |     |    |    |    |     | 24 | 26* | 28 |
|   |   |   |   |   |    |    |     |    |    |    |     |    | 24* |    |

25 not found

Enter a number between 0 and 28: 8

Subscripts:

| 0 | 1 | 2 | 3  | 4 | 5   | 6  | 7   | 8  | 9  | 10 | 11 | 12 | 13 | 14 |
|---|---|---|----|---|-----|----|-----|----|----|----|----|----|----|----|
| 0 | 2 | 4 | 6  | 8 | 10  | 12 | 14* | 16 | 18 | 20 | 22 | 24 | 26 | 28 |
| 0 | 2 | 4 | 6* | 8 | 10  | 12 |     |    |    |    |    |    |    |    |
|   |   |   |    | 8 | 10* | 12 |     |    |    |    |    |    |    |    |
|   |   |   |    |   | 8*  |    |     |    |    |    |    |    |    |    |

8 found in array element 4

Enter a number between 0 and 28: 6

Subscripts:

| 0 | 1 | 2 | 3  | 4 | 5  | 6  | 7   | 8  | 9  | 10 | 11 | 12 | 13 | 14 |
|---|---|---|----|---|----|----|-----|----|----|----|----|----|----|----|
| 0 | 2 | 4 | 6  | 8 | 10 | 12 | 14* | 16 | 18 | 20 | 22 | 24 | 26 | 28 |
| 0 | 2 | 4 | 6* | 8 | 10 | 12 |     |    |    |    |    |    |    |    |

6 found in array element 3

图 4.20 排序数组的折半查找过程

## 4.9 多下标数组

C++中有多下标数组。多下标数组常用于表示由行和列组成的表格。要表示表格中的元素，就要指定两个下标：习惯上第一个表示元素的行，第二个表示元素的列。

用两个下标表示特定的表格或数组称为双下标数组 (double-subscripted array)。注意，多下标数组可以有多于两个的下标，C++编译器支持至少12个数组下标。图4.21演示了双下标数组a。数组包含三行四列，因此是3×4数组。一般来说，m行和n列的数组称为m×n数组。

数组a中的每个元素如图4.21所示，元素名字表示为a[i][j]，a是数组名，i和j是惟一标识a中每个元素的下标。注意第一行元素名的第一个下标均为0，第四列的元素名的第二个下标均为3。

## 常见编程错误 4.11

把双下标数组元素  $a[x][y]$  误写成  $a[x, y]$ 。实际上,  $a[x, y]$  等于  $a[y]$ , 因为 C++ 将包含逗号运算符的表达式  $(x, y)$  求值为  $y$  (逗号分隔的表达式中最后一个项目)。

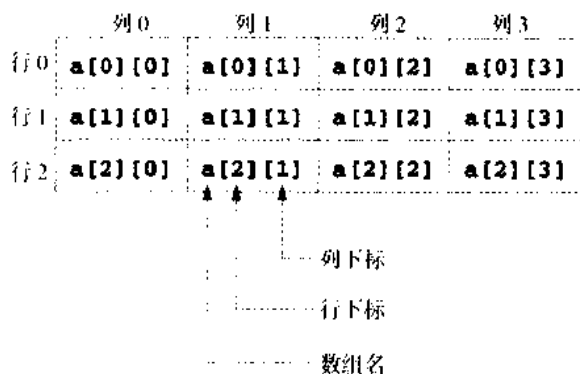


图 4.21 三行四列的双下标数组

多下标数组可以在声明中初始化, 就像单下标数组一样。例如, 双下标数组  $b[2][2]$  可以声明和初始化如下:

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

数值用花括号按行分组, 因此 1 和 2 初始化  $b[0][0]$  和  $b[0][1]$ , 3 和 4 初始化  $b[1][0]$  和  $b[1][1]$ 。如果指定行没有足够的初始化值, 则该行的其余元素初始化为 0。这样, 下列声明:

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

初始化  $b[0][0]$  为 1、 $b[0][1]$  为 0、 $b[1][0]$  为 3、 $b[1][1]$  为 4。

图 4.22 演示了声明中初始化双下标数组。程序声明三个数组, 各有两行三列。array1 的声明在两个子列表中提供六个初始化值。第一个子列表初始化数组第一行为 1、2、3, 第二个子列表初始化数组第二行为 4、5、6。如果从 array1 初始化值列表中删除每个子列表的花括号, 则编译器自动先初始化第一行元素, 然后初始化第二行元素。

array2 声明提供六个初始化值。初始化值先赋给第一行, 再赋给第二行。任何没有显式初始化值的元素都自动初始化为 0, 因此  $array2[1][2]$  自动初始化为 0。

```
1 // Fig. 4.22: fig04_22.cpp
2 // Initializing multidimensional arrays
3 #include <iostream.h>
4
5 void printArray( int[][ 3 ] );
6
7 int main()
8 {
9     int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } },
10        array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 },
11        array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
12
13     cout << "Values in array1 by row are:" << endl;
14     printArray( array1 );
15
16     cout << "Values in array2 by row are:" << endl;
17     printArray( array2 );
```

```

18
19  cout << "Values in array3 by row are:" << endl;
20  printArray( array3 );
21
22  return 0;
23 }
24
25 void printArray( int a[][ 3 ] )
26 {
27     for ( int i = 0; i < 2; i++ ) {
28
29         for ( int j = 0; j < 3; j++ )
30             cout << a[ i ][ j ] << ' ';
31
32         cout << endl;
33     }
34 }

```

**输出结果:**

```

Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0

```

图 4.22 初始化多维数组

array3 的声明在两个子列表中提供 3 个初始化。第一行的子列表显式地将第一行的前两个元素初始化为 1 和 2，第 3 个元素自动初始化为 0。第二行的子列表显式地将第一个元素初始化为 4，最后两个元素自动初始化为 0。

程序调用函数 printArray 输出每个数组的元素。注意函数定义指定数组参数为 int a[ ][ 3 ]。函数参数中收到单下标数组时，函数参数表中的数组括号是空的。多下标数组第一个下标的长度也不需要，但随后的所有下标长度都是必须的。编译器用这些长度确定多下标数组中元素在内存中的地址。不管下标数是多少，所有数组元素都是在内存中顺序存放。在双下标数组中，第一行后面的内存地址存放第二行。

在参数声明中提供下标使编译器能告诉函数如何找到数组中的元素。在双下标数组中，每行是一个单下标数组。要找到特定行中的元素，函数要知道每一行有多少元素，以便在访问数组时跳过适当数量的内存地址。这样，访问 a[ 1 ][ 2 ] 时，函数知道跳过内存中第一行的 3 个元素以访问第二行（行 1），然后访问这一行的第 3 个元素（元素 2）。

许多常见数组操作使用 for 重复结构。例如，下列 for 结构（见图 4.21）将数组 a 第三行的所有元素设置为 0：

```

for ( column = 0; column < 4; column++ )
    a[ 2 ][ column ] = 0;

```

我们指定第三行，因此知道第一个下标总是 2（0 是第一行的下标，1 是第二行的下标）。for 循环只改变第二个下标（即列下标）。上述 for 结构等同于下列赋值语句：



```

a [ 2 ] [ 0 ] = 0;
a [ 2 ] [ 1 ] = 0;
a [ 2 ] [ 2 ] = 0;
a [ 2 ] [ 3 ] = 0;

```

下列嵌套 for 结构确定数组 a 中所有元素的总和:

```

total = 0;

for ( row = 0; row < 3; row++ )
    for ( column = 0; column < 4; column++ )
        total += a [ row ] [ column ];

```

for 结构一次一行地求数组中所有元素的和。外层 for 结构首先设置 row (即行下标) 为 0, 使第一行的元素可以用内层 for 结构求和。然后外层 for 结构将 row 递增为 1, 使第二行的元素可以用内层 for 结构求和。然后外层 for 结构将 row 递增为 2, 使第三行的元素可以用内层 for 结构求和。结束嵌套 for 结构之后, 打印结果。

图 4.23 的程序对  $3 \times 4$  数组 studentGrades 进行几个其他常见数组操作。每行数组表示一个学生, 每列表示学生期末考试中 4 门成绩中的一门成绩。数组操作用 4 个函数进行。函数 minimum 确定该学期所有学生考试成绩中的最低成绩。函数 maximum 确定该学期所有学生考试成绩中的最高成绩。函数 average 确定每个学生该学期的平均成绩。函数 printArray 以整齐的表格形式输出双下标数组。

```

1 // Fig. 4.23: fig04_23.cpp
2 // Double-subscripted array example
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 const int students = 3;    // number of students
7 const int exams = 4;      // number of exams
8
9 int minimum( int [][] exams, int, int );
10 int maximum( int [][] exams, int, int );
11 float average( int [], int );
12 void printArray( int [][] exams, int, int );
13
14 int main()
15 {
16     int studentGrades[ students ][ exams ] =
17         { { 77, 68, 86, 73 },
18           { 96, 87, 89, 78 },
19           { 70, 90, 86, 81 } };
20
21     cout << "The array is:\n";
22     printArray( studentGrades, students, exams );
23     cout << "\n\nLowest grade: "
24         << minimum( studentGrades, students, exams )
25         << "\nHighest grade: "
26         << maximum( studentGrades, students, exams ) << '\n';
27
28     for ( int person = 0; person < students; person++ )
29         cout << "The average grade for student " << person << " is "
30             << setiosflags( ios::fixed | ios::showpoint )
31             << setprecision( 2 )

```

```
32         << average( studentGrades[ person ], exams ) << endl;
33
34     return 0;
35 }
36
37 // Find the minimum grade
38 int minimum( int grades[][ exams ], int pupils, int tests )
39 {
40     int lowGrade = 100;
41
42     for ( int i = 0; i < pupils; i++ )
43
44         for ( int j = 0; j < tests; j++ )
45
46             if ( grades[ i ][ j ] < lowGrade )
47                 lowGrade = grades[ i ][ j ];
48
49     return lowGrade;
50 }
51
52 // Find the maximum grade
53 int maximum( int grades[][ exams ], int pupils, int tests )
54 {
55     int highGrade = 0;
56
57     for ( int i = 0; i < pupils; i++ )
58
59         for ( int j = 0; j < tests; j++ )
60
61             if ( grades[ i ][ j ] > highGrade )
62                 highGrade = grades[ i ][ j ];
63
64     return highGrade;
65 }
66
67 // Determine the average grade for a particular student
68 float average( int setOfGrades[], int tests )
69 {
70     int total = 0;
71
72     for ( int i = 0; i < tests; i++ )
73         total += setOfGrades[ i ];
74
75     return ( float ) total / tests;
76 }
77
78 // Print the array
79 void printArray( int grades[][ exams ], int pupils, int tests )
80 {
81     cout << "          [ 0 ] [ 1 ] [ 2 ] [ 3 ]";
82
83     for ( int i = 0; i < pupils; i++ ) {
84         cout << "\nstudentGrades[ " << i << " ] ";
85
86         for ( int j = 0; j < tests; j++ )
```

```

87         cout << setiosflags( ios::left ) << setw( 5 )
88         << grades[ i ][ j ];
89     }
90 }

```

**输出结果:**

```

The array is:
           [ 0 ] [ 1 ] [ 2 ] [ 3 ]
StudentGrades[ 0 ]  77  68  86  73
StudentGrades[ 1 ]  96  87  98  78
StudentGrades[ 2 ]  70  90  86  81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75

```

图 4.23 使用双下标数组举例

函数 `minimum`、`maximum` 和 `printArray` 取得三个参数——`studentGrades` 数组（在每个函数中调用 `grades`）、学生数（数组行）、考试门数（数组列）。每个函数用嵌套 `for` 结构对 `grades` 数组循环。下列嵌套 `for` 结构来自函数 `minimum` 的定义：

```

for ( i = 0; i < pupils; i++ )
    for ( j = 0; j < tests; j++ )
        if ( grades [ i ][ j ] < lowGrade )
            lowGrade = grades [ i ][ j ];

```

外层 `for` 结构首先将 `i`（行下标）设置为 0，使第一行的元素可以和内层 `for` 结构体中的变量 `lowGrade` 比较。内层 `for` 结构比较 `lowGrade` 与 4 个成绩中的每一门成绩。如果成绩小于 `lowGrade`，则 `lowGrade` 设置为该门成绩。然后外层 `for` 结构将行下标加到 1，第二行的元素与变量 `lowGrade` 比较。接着外层 `for` 结构行下标加到 2，第三行的元素与变量 `lowGrade` 比较。嵌套 `for` 结构执行完成后，`lowGrade` 包含双下标数组中的最小值。函数 `maximum` 的执行过程与函数 `minimum` 相似。

函数 `average` 取两个参数：一个单下标数组为某个学生的考试成绩，一个数字表示数组中有几个考试成绩。调用 `average` 时，第一个参数 `studentGrades[ student ]` 指定将双下标数组 `studentGrades` 的特定行传递给 `average`。例如，参数 `studentGrades[ 1 ]` 表示双下标数组 `studentGrades` 第二行中存放的 4 个值（成绩的单下标数组）。双下标数组可以看作元素是单下标数组的数组。函数 `average` 计算数组元素的和，将总和除以考试成绩的个数，并返回一个浮点数结果。

## 4.10 有关对象的思考：确定类的行为

第 2 章和第 3 章“有关对象的思考”一节完成了电梯模拟程序面向对象设计的前两步，即确定实现电梯模拟程序的类和这些类的属性。

本节要确定实现电梯模拟程序所需的类行为。第 5 章将介绍这些类对象之间的交互。

下面要考虑一些实际类对象的行为。收音机的行为包括可以选台和设置音量。汽车的行为包括加速（按油门板）和减速（按制动闸）。

可以看出,对象通常不是自动做出行为,而是在向对象发出消息(message)时调用特定行为,这个消息表示对象要做出特定行为。这很像函数调用,C++中就是这样向对象发出消息的。

### 电梯实验室任务3

1. 继续处理第3章生成的事实文件。其中将文件分成两组事实,第一组是属性,第二组是其他事实。
2. 对每个数,加进第三个组,称为“行为”。用该组存放类的每个行为,对象调用这些行为来做某件事(即通过向对象发出消息)。例如,人可以单击按钮,因此将pushButton列为按钮类的行为。函数pushButton和按钮类的其他行为称为成员函数(member function)或方法(method)。类的属性(如按钮“开”或“关”)通常称为按钮类的数据成员(data member)。类的成员函数通常操作类的数据成员(如pushButton改变按钮的属性为“开”)。成员函数通常还向其他类的对象发送消息(例如按钮对象向电梯发送comeGetMe消息)。假设某人按按钮时电梯的按钮亮灯,电梯到达某一层时,电梯要发送resetButton消息以关掉按钮的灯。电梯可能要确定某个按钮有没有按下,因此要用getButton行为检查按钮,返回1和0分别表示按钮“开”或“关”。可能要让电梯门响应消息openDoor和closeDoor等等。
3. 对类指定的每个行为,简要介绍这个行为的作用。列出行为导致的属性改变,列出行为向其他类的对象发送的消息。

### 说明

1. 首先列出问题陈述中提到的类属性,然后列出问题陈述中隐含的类属性。
2. 在需要时增加相应的行为。
3. 系统设计不是完善和完整的过程,只要尽力而为,后面几章的练习会介绍如何修改。
4. 这个阶段很难看出所有行为,第5章练习中可能还要增加新的行为。

### 小结

- C++ 将数值清单存放在数组中。数组是相同名称和相同类型的一组连续内存地址。要引用数组中的特定位置或元素,就要指定数组中的特定位置或元素的位置号。
- 下标应为整数或整型表达式。如果程序用整型表达式下标,则要求值这个整型表达式以确定下标。
- 一定要注意“数组第7个元素”与“数组元素7”之间的差别。由于数组下标从0开始,因此“数组第7个元素”的下标为6,而“数组元素7”的下标为7,是第8个元素。这常常是“差1错误”的原因。
- 数组要占用内存空间。下列声明对整型数组b保留100个元素,对整型数组x保留27个元素:  

```
int b[ 100 ], x[ 27 ];
```
- char 类型的数组可以存放字符串。
- 数组元素可以用声明、赋值或输入初始化。
- 如果初始化值比数组中的元素少,则其余元素自动初始化为0。
- C++ 无法阻止计算机引用超出数组边界的元素。
- 可以用字符串直接初始化字符数组。
- 所有字符串均用空字符结尾(‘\0’)。

- 字符数组还可以用初始化值列表中的各个字符常量初始化。
- 可以用数组下标符号直接访问字符串中的各个字符。
- 将数组传递给函数实际上传递的是数组名, 要将数组中的单个元素传递给函数, 只需传递数组名和指定元素的下标 (放在方括号中) 即可。
- C++ 模拟按引用调用, 自动将数组传递给函数, 被调用函数可以修改调用者原数组中的元素值。数组名的值为数组中第一个元素的地址。由于传递数组的开始地址, 因此被调用函数知道数组的准确存放位置。
- 要让函数通过函数调用接收数组, 函数的参数表应指定接收数组。数组方括号中的数组长度不是必需的。
- 有时程序中函数不能修改数组元素。由于数组总是模拟按引用调用, 因此数组中数值的修改很难控制。C++ 提供类型限定符 `const`, 可以防止修改函数中的数组值。数组参数前面加上 `const` 限定符时, 数组元素成为函数体中的常量, 要在函数体中修改数组元素会造成语法错误。
- 数组可以用冒泡方法排序。这个方法在数组中多次操作, 每一次都比较一对相邻元素。如果某一对为升序 (或数值相等), 数值保持不变。如果某一对为降序, 则将数值交换。对于小数组, 可以用冒泡方法排序, 但对于大数组, 则不如用更复杂的排序方法那么有效。
- 线性查找比较数组中每个元素与查找键。由于数组没有特定的顺序, 很可能第一个就找到, 也可能要到最后一个才找到。因此, 平均起来, 程序要比较数组中一半的元素才能找到查找键值。线性查找方法适用于小数组或未排序数组。
- 折半查找算法在每次比较之后排除所查找数组的一半。这个算法找到数组的中间位置, 将其与查找键比较。如果相等, 则已找到查找键, 返回该元素的数组下标。否则将问题简化为查找一半数组。
- 在最糟糕的情况下, 查找 1024 个元素的数组只要用折半查找进行十次比较。
- C++ 中有多下标数组。多下标数组常用于表示由行和列组成的表格。要表示特定表格元素, 就要指定两个下标: 习惯上第一个表示元素的行, 第二个表示元素的列。用两个下标表示特定的表格或数组称为双下标数组。
- 函数参数中收到单下标数组时, 函数参数表中的数组方括号是空的。多下标数组第一个下标的长度也不需要, 但后续所有下标长度都是需要的。编译器用这些长度确定多下标数组中元素在内存中的地址。
- 要将双下标数组的一行传递给函数而函数只接收单下标数组, 只要传递数组名加上第一个下标即可。

## 术语

|                                  |                                               |
|----------------------------------|-----------------------------------------------|
| <code>a[i]</code>                | column subscript 列下标                          |
| <code>a[i][j]</code>             | const type qualifier <code>const</code> 类型限定符 |
| array 数组                         | constant variable 常量变量                        |
| array initializer list 数组初始化值列表  | declare an array 声明数组                         |
| binary search of an array 数组折半查找 | double-subscripted array 双下标数组                |
| bounds checking 边界检查             | element of an array 数组元素                      |
| bubble sort 冒泡排序                 | initialize an array 初始化数组                     |

|                                      |                                              |
|--------------------------------------|----------------------------------------------|
| initializer 数组的初始化值                  | search an array 查找数组                         |
| linear search of an array 数组的线性查找    | search key 查找键                               |
| magic number 魔数                      | simulated call-by-reference 模拟的按引用调用         |
| m-by-n array $m \times n$ 数组         | single-subscripted array 单下标数组               |
| multiple-subscripted array 多下标数组     | sinking sort 下沉排序                            |
| name of an array 数组名                 | sort an array 数组排序                           |
| named constant 命名常量                  | square brackets 方括号 ([])                     |
| null character 空字符 ('\\0')           | string 字符串                                   |
| off-by-one error 差 1 错误              | subscript 下标                                 |
| pass-by-reference 按引用传递              | table of values 数值表                          |
| pass of a bubble sort 冒泡排序操作         | tabular format 表格形式                          |
| passing arrays to functions 将数组传递给函数 | temporary area of exchange of values 临时数据交换区 |
| position number 位置号                  | triple-subscripted array 三下标数组               |
| row subscript 行下标                    | value of an element 元素值                      |
| scalability 伸缩性                      | "walk off" an array 数组超界                     |
| scalar 标量                            | zeroth element 第 0 个元素                       |

## 自测练习

### 4.1 填空：

- 清单和表格值存放在 \_\_\_\_\_ 中。
- 数组元素的关系是 \_\_\_\_\_ 和 \_\_\_\_\_ 相同。
- 用于引用数组中特定元素的数字称为数组的 \_\_\_\_\_。
- \_\_\_\_\_ 用于声明数组长度，使程序伸缩性更强。
- 将数组元素按顺序排列的过程称为数组 \_\_\_\_\_。
- 确定数组中是否包含某个键值的过程称为数组 \_\_\_\_\_。
- 使用两个下标的数组称为 \_\_\_\_\_ 数组。

### 4.2 判断下列各题是否正确。如果不正确，请说明原因。

- 数组可以存放许多不同类型的数值。
- 数组下标通常为 float 数据类型。
- 如果初始化值列表中的初始化值少于数组元素个数，则其余元素自动初始化为初始化值列表中的最后一个值。
- 初始化值列表中的初始化值多于数组元素个数是个错误。
- 将单个数组元素传给函数并修改该元素值，那么在被调用函数执行结束时仍保留修改后的值。

### 4.3 回答下列关于数组 fractions 的问题：

- 定义常变量 arraySize，初始化为 10。
- 声明数组的 arraySize 元素类型为 float，并将元素初始化为 0。
- 数组开头第 4 个元素的名称。
- 引用数组元素 4。

- e) 将数值 1.667 赋给数组元素 9。
- f) 将 3.333 赋给数组第 7 个元素。
- g) 打印数组元素 6 和 9, 小数点后面为两位精度, 并显示屏幕输出。
- h) 用 for 重复结构打印数组的所有元素。定义整型变量 x 为循环控制变量。显示输出。

#### 4.4 回答下列关于数组 table 的问题:

- a) 声明 3 行 3 列的整型数组。假设定义常量变量 arraySize 为 3。
- b) 数组包含多少元素。
- c) 用 for 重复结构初始化数组每个元素为该元素下标的和。假设声明整型变量 x 和 y 为循环控制变量。
- d) 编写一个程序段, 以 3 行 3 列的表格形式打印每个数组元素值。假设用下列声明初始化数组:

```
int table[ arraySize ][ arraySize ] = { { 1, 8 }, { 2, 4, 6 }, { 5 } };
```

并声明整型变量 x 和 y 为循环控制变量。显示输出。

#### 4.5 找出下列语句中的错误并说明如何纠正。

- a) `#include <iostream.h>;`
- b) `arraySize = 10; // arraySize was declared const`
- c) 假设 `int b[ 10 ] = { 0 };`  
`for ( int i = 0; i <= 10; i++ )`  
`b[ i ] = 1;`
- d) 假设 `int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`  
`a[ 1, 1 ] = 5;`

### 自测练习答案

- 4.1 a) 数组。b) 名称、类型。c) 下标。d) 常量变量。e) 排序。f) 查找。g) 双下标。
- 4.2 a) 不正确。任何数组只能存放相同类型的值。
- b) 不正确。数组下标通常为整数或整型表达式。
- c) 不正确。其余元素自动初始化为 0。
- d) 正确。
- e) 不正确。数组各个元素通过按值调用传递。如果将整个数组传递给函数, 则任何修改将影响原始数组。
- 4.3 a) `const int arraySize = 10;`
- b) `float fractions[ arraySize ] = { 0 };`
- c) `fractions[ 3 ]`
- d) `fractions[ 4 ]`
- e) `fractions[ 9 ] = 1.667;`
- f) `fractions[ 6 ] = 3.333;`
- g) `cout << setiosflags ( ios::fixed | ios::showpoint )`  
`<< setprecision( 2 ) << fractions[ 6 ] << ' '`  
`<< fractions[ 9 ] << endl;`  
 输出:  
 3.33 1.67

```
h) for ( int x = 0; x < arraySize; x++ )
    cout << "fractions[ " << x << " ] = " << fractions[ x ]
        << endl;
```

输出:

```
fractions[ 0 ] = 0
fractions[ 1 ] = 0
fractions[ 2 ] = 0
fractions[ 3 ] = 0
fractions[ 4 ] = 0
fractions[ 5 ] = 0
fractions[ 6 ] = 3.333
fractions[ 7 ] = 0
fractions[ 8 ] = 0
fractions[ 9 ] = 1.667
```

4.4 a) `int table[ arraySize ] [ arraySize ];`

b) 9个.

```
c) for ( x = 0; x < arraySize; x++ )
    for ( y = 0; y < arraySize; y++ )
        table[ x ][ y ] = x + y;
```

```
d) cout << "    [ 0 ]    [ 1 ]    [ 2 ]" << endl;
    for ( int x = 0; x < arraySize; x++ ) {
        cout << "[ " << x << " ]";
        for ( int y = 0; y < arraySize; y++ )
            cout << setw(3) << table[ x ][ y ] << " ";
        cout << endl;
```

输出:

```
    [ 0 ]    [ 1 ]    [ 2 ]
[ 0 ]    1      8      0
[ 1 ]    2      4      6
[ 2 ]    5      0      0
```

4.5 a) 不正确: `#include` 预处理指令后面不应有分号。

纠正: 删除 `#include` 预处理指令后面的分号。

b) 不正确: 用赋值语句对常量变量赋值。

纠正: 在 `const int arraySize` 声明中对常量变量赋值。

c) 不正确。在数组边界之外引用数组元素(`b[ 10 ]`)。

纠正: 将控制变量的终值变为9。

d) 不正确: 数组下标错误。

纠正: 将语句变为 `a[ 1 ][ 1 ] = 5;`

## 练习

4.6 填空:

- C++ 在 \_\_\_\_\_ 中存放数值清单。
- 数组元素的关系是 \_\_\_\_\_。
- 引用数组元素时, 括号中包含的位置号称为 \_\_\_\_\_。
- 数组 `p` 的4个元素名为 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- 命名数组、指定数组类型和指定数组中的元素个数称为数组 \_\_\_\_\_。
- 将数组元素按升序或降序排列的过程称为 \_\_\_\_\_。



- g) 在双下标数组中, 习惯上第一个下标表示数组元素的 \_\_\_\_\_, 第二个下标表示数组元素的 \_\_\_\_\_。
- h)  $m \times n$  数组包含 \_\_\_\_\_ 行、\_\_\_\_\_ 列和 \_\_\_\_\_ 个元素。
- i) 数组 d 中行 3 列 5 的元素名为 \_\_\_\_\_。
- 4.7 判断下列各题是否正确。如果不正确, 请说明原因。
- a) 要引用数组中的特定位置或元素, 我们指定数组名和特定元素的值。
- b) 数组声明为这个数组保留内存空间。
- c) 要为整型数组 p 保留 100 个位置, 声明如下:
- ```
pl 100 ];
```
- d) 将 15 个元素的数组中的元素初始化为 0, C++ 程序至少要包含一个 for 循环。
- e) 求双下标数组元素和, C++ 程序要包含嵌套 for 结构。
- 4.8 编写完成下列任务的 C++ 语句:
- a) 显示字符数组 f 第七个元素的值。
- b) 将数值输入单下标浮点数数组 b 的元素 4。
- c) 将单下标整型数组 g 的 5 个元素各初始化为 8。
- d) 求出浮点数数组 c 的 100 个元素之和并打印这些元素。
- e) 将 a 数组复制到数组 b 的开头部分。假设 "float a[ 11 ], b[ 34 ];"
- f) 确定和打印 99 个元素的浮点数数组 w 中的最小值和最大值。
- 4.9 考虑  $2 \times 3$  整型数组 t, 并编写相应的语句。
- a) 编写 t 的声明。
- b) t 有多少行?
- c) t 有多少列?
- d) t 有多少个元素?
- e) 写出 t 的第 2 行所有元素名。
- f) 写出 t 的第 3 列所有元素名。
- g) 将 t 中行 1 列 2 的元素设置为 0。
- h) 将 t 的每个元素初始化为 0, 不用重复结构。
- i) 用嵌套 for 结构将 t 的每个元素初始化为 0。
- j) 从终端输入 t 的元素值。
- k) 用一系列语句确定和打印数组 t 的最小值。
- l) 显示 t 的第 1 行元素。
- m) 显示 t 的第 4 列元素。
- n) 以整齐的表格形式打印数组 t, 将列下标设为输出首部的一行, 行下标放在输出左部的一列。
- 4.10 用单下标数组解决下列问题。公司按佣金为员工发工资, 销售员每周发 200 美元加上本周总销售额的 9%。例如, 如果某个销售员本周总销售额为 5000 美元, 则发  $200 + 9\% \times 5000 = 650$  美元。编写一个程序 (用计数器数组) 确定工资在下列范围的员工数 (假设将每个销售人员的工资取整):
- a) \$200~\$299
- b) \$300~\$399

- c) \$400-\$499
- d) \$500-\$599
- e) \$600-\$699
- f) \$700-\$799
- g) \$800-\$899
- h) \$900-\$999
- i) \$1000 以上

4.11 图4.16的冒泡排序方法在大数组中不适用,通过下列简单修改可以提高冒泡排序方法的性能。

- a) 第一遍之后,最大数总是数组中编号最大的元素,第二遍之后,次大数总是数组中编号次大的元素等等。不用每次都作九次比较,而只要第二遍比较8次,第三遍比较七次等等。
- b) 数组中的数据可能已经有正确或接近正确的顺序,为什么一定要进行九遍比较呢?每遍之后检查是否有所交换。如果没有交换,则数据已经正确排序,程序终止。如果有交换,则至少还要再进行一遍。

4.12 编写一条语句,完成下列单下标数组操作:

- a) 将整型数组 counts 的 10 个元素初始化为 0。
- b) 将整型数组 bonus 的 15 个元素分别加 1。
- c) 从键盘读取 float 数组 monthlyTemperatures 的 12 个值。
- d) 用列格式打印整型数组 bestScores 的 5 个值。

4.13 寻找下列语句中的错误。

- a) 假设: `char str[ 5 ] ;`  
`cin >> str; // User types hello`
- b) 假设: `int a[ 3 ] ;`  
`cout << a[ 1 ] << " " << a[ 2 ] << " " << a[ 3 ] << endl;`
- c) `float f[ 3 ] = { 1.1, 10.01, 100.001, 1000.0001 };`
- d) 假设: `double d[ 2 ][ 10 ] ;`  
`d[ 1, 9 ] = 2.345;`

4.14 修改图 4.17 的程序,使函数 mode 能处理模值的连接。并修改函数 median,使偶数个元素的数组中 median 为两个中间元素的平均值。

4.15 用单下标数组解决下列问题。读取 20 个数,各为 10 到 100 之间的值。读取每个数时,只打印不重复的值。“最糟的情况”是 20 个值都不重复。用最小的数组解决这个问题。

4.16 打印  $3 \times 5$  双下标数组 sales 的元素,表示下列语句中将其设置为 0 的顺序:

```
for ( row = 0; row < 3; row++ )
    for ( column = 0; column < 5; column++ )
        sales[ row ] [ column ] = 0;
```

4.17 编写一个程序，模拟投两个骰子。程序用rand函数投第一个骰子，并再次用rand函数投第二个骰子，然后计算两个值的和。说明：由于每个骰子显示1到6的整数值，因此两个骰子的和为2到12，7最常见，2和12最不常见。图4.24显示了36种可能的两个骰子的和。程序将投两个骰子36 000次，用单下标数组估算每个和出现的次数，用表格形式打印结果。并确定和是否合理，即有六种方式投出7，因此有六分之一的可能投出7。

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

图 4.24 36种可能的两个骰子的和

4.18 下列程序有什么作用？

```

1 // ex04_18.cpp
2 #include <iostream.h>
3
4 int whatIsThis( int [], int );
5
6 int main()
7 {
8     const int arraySize = 10;
9     int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10
11     int result = whatIsThis( a, arraySize );
12
13     cout << "Result is" << result << endl;
14     return 0;
15 }
16
17 int whatIsThis( int b[], int size )
18 {
19     if ( size == 1 )
20         return b[ 0 ];
21     else
22         return b[ size - 1 ] + whatIsThis(b, size - 1 );
23 }

```

4.19 编写一个程序，运行1000次投骰子游戏，并回答下列问题：

- 第一次、第二次、...第十二次和第十二次以后赢了几场？
- 第一次、第二次、...第十二次和第十二次以后输了几场？
- 赢的机会会有多少（投骰子游戏是最公平的游戏之一，为什么）？
- 投骰子游戏平均长度是多少？
- 游戏玩得越久，赢的机会是否越多？

4.20 (航空订票系统) 小航空公司购买了一台用于航空订票系统的计算机, 要求对新系统编程, 对每个航班订座 (每班 10 位)。

程序显示下列菜单选项: Please type 1 for "smoking" 和 please type 2 for "nonsmoking"。如果输入 1, 则程序指定吸烟舱位 (座位 1 到 5), 如果输入 2, 则程序指定非吸烟舱位 (座位 6 到 10)。程序应打印一个登机牌, 表示座位号和是否为吸烟舱位。

用一个单下标数组表示飞机的座位图。将数组的所有元素初始化为 0, 表示所有座位都是空的。订每个座位时, 将数组相应元素设置为 1, 表示该座位已订。

当然, 程序不能再订已经订过的座位。吸烟舱位已满时, 应询问可否订非吸烟舱位; 同样, 非吸烟舱位已满时, 应询问可否订吸烟舱位。如果同意, 再相应订座, 否则打印消息 "Next flight leaves in 3 hours."。

4.21 下列程序有什么作用?

```
1 // ex04_21.cpp
2 #include <iostream.h>
3
4 void someFunction( int [], int );
5
6 int main()
7 {
8     const int arraySize = 10;
9     int a[ arraySize ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11     count << "The values in the array are:" << endl;
12     someFunction( a, arraySize );
13     cout << endl;
14     return 0;
15 }
16
17 void someFunction( int b[], int size )
18 {
19     if ( size > 0 ) {
20         someFunction( &b[ 1 ], size - 1 );
21         cout << b[ 0 ] << " ";
22     }
23 }
```

4.22 用双下标数组解决下列问题。公司有 4 个销售员 (1 到 4), 销售五种产品 (1 到 5)。每天每个销售人员要报告每种产品的销售量, 报告中包含:

- a) 销售员号
- b) 产品号
- c) 当日总销售额

这样, 每个销售人员每天要送 0 到 5 份报告。假设上个月的这些信息都有, 编写一个程序, 读取上月销售情况的信息, 并按每个销售人员和每种产品进行汇总。所有汇总存放在双下标数组 sales 中。处理上个月的所有信息后, 将结果打印成表格形式, 每列表示一个销售员, 每行表示一种产品。通过交叉求和求出上月每个销售人员的总销售额和每种产品的总销售额。表格输出的右边小计行和下边小计列中应打印这些信息。

4.23 (龟图) Logo 语言在个人计算机用户中非常流行, 该语言形成了龟图的概念。假设有个机器海龟, 通过 C++ 程序控制在房子中移动。在两个方向之一打开画笔, 即向上或向下。

画笔向下时,海龟跟踪移动的形狀并留下移动的路径,画笔向上时,海龟自由移动,不写下任何东西。在这个问题中,要模拟海龟的操作和生成计算机化的草图框。

用  $20 \times 20$  数组 floor, 初始化为 0。从数组中读取命令。跟踪任何时候海龟的当前位置和画笔的向上或向下状态。假设海龟总是从位置 0, 0 开始, 画笔向上。程序要处理的海龟命令如下:

命令	含义
1	笔向上
2	笔向下
3	右转
4	左转
5, 10	前进 10 格 (或几格)
6	打印 $20 \times 20$ 数组
9	数据结束 (标记)

假设海龟接近平面中心。下列“程序”绘制和打印  $12 \times 12$  正方形并让画笔向上:

```
2
5, 12
3
5, 12
3
5, 12
3
5, 12
1
6
9
```

画笔向下并移动海龟时, 将数组 floor 的相应元素设置为 1。指定命令 6 (打印) 时, 只要数组中有 1, 就显示星号或选择的其他符号, 而 0 则显示空白。编写一个程序, 实现这里介绍的龟图功能。编写几个龟图程序, 画一些有趣的图形。增加其他命令以增加龟图语言的功能。

- 4.24 (骑士旅行) 国际象棋中, 最有趣的问题之一是骑士旅行问题, 最初是由数学家欧拉提出的。问题如下: 能否让骑士在空棋盘上移动, 在 64 个方格中经过且只经过一次? 下面要详细介绍这个问题。

骑士的移动按 L 形路线 (一个方向两格, 另一垂直方向一格, 相当于“马走日”)。这样, 从空棋盘中央的方格中, 骑士可以有 8 种不同的移动方向 (编号为 0 到 7), 如图 4.25。

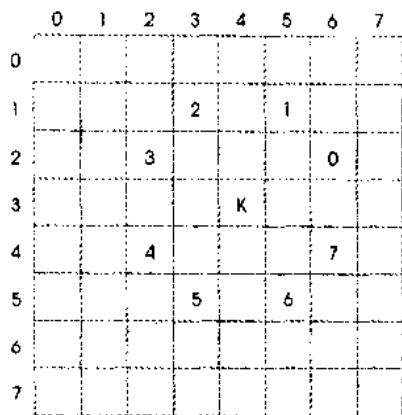


图 4.25 骑士有 8 种不同的移动方向

- a) 在纸上画一个  $8 \times 8$  棋盘, 试用手工移动骑士, 在移动的第一个框中标上 1、第二个框中标上 2、第三个框中标上 3 等等。事先要估计走多远, 总共有 64 步, 能走多远?
- b) 然后要开发一个在棋盘上移动骑士的程序。棋盘用  $8 \times 8$  的双下标数组 `board` 表示。每个格子初始化为 0。我们用水平和垂直组件描述 8 种可能的移动方式。例如, 图 4.25 中 0 类型的移动是水平右移两格和垂直上移一格。2 类型的移动是水平左移一格和垂直上移两格。左移和上移表示为负数。8 种移动可以用两个单下标数组 `horizontal` 和 `vertical` 描述如下:

```
horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 2 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1
```

让变量 `currentRow` 和 `currentColumn` 表示骑士当前位置的行和列, 要进行 `moveNumber` 类型的移动 (`moveNumber` 在 0 到 7 之间), 程序用下列语句:

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

计数器应在 1 到 64 之间改变, 用来记录每一格中骑士最近一次移动的顺序号。记住, 要测试各种可能的移动, 确定骑士是否已访问过该格。当然, 要测试各种可能的移动, 保证骑士不会跳到棋盘以外。现在编写在棋盘上移动骑士的程序。运行程序, 看看骑士移动了多少位?

- c) 编写和运行骑士旅行程序之后, 可以进行一些透试, 我们要开发一个移动骑士的试探程序 (或策略)。试探不一定成功, 但认真设计的试探方法能提高成功的机会。可以发现, 外层格子比在棋盘中心附近的格子更难移动。事实上, 最难访问的是四角的格子。

凭直觉, 应先将骑士移到最难到达的格子, 在旅行即将结束时再访问最容易到达的格子, 这样成功的机率较高。

我们可以开发一个访问性试探, 将每格进行访问性分类, 然后总是设法把骑士移到最难访问的点。我们在双下标数组 `accessibility` 中标上每个格能访问的格数。在空棋盘上, 中间格的值为 8, 四角格的值为 2, 其他格的值为 3、4、6, 如下所示:

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

现在用访问性试探值编写骑士旅程序。任何时候，总是设法把骑士移到最难访问的点。对于格子连接的情况，骑士可能移到任一连接的格子。因此，旅行可以从四角开始（说明：骑士在棋盘上移动时，程序随着越来越多的格子被占而减少访问性值。这样，任何时候，棋盘上任一格子上访问性值是该格子能访问的格数）。运行这个程序，能否一一访问棋盘上的格子？现在修改程序，从棋盘上任何一格开始运行64次旅行，哪些能全部访问？

- d) 编写一个骑士旅程序，在遇到两格或几格连接时，确定选哪一格时先考虑从连接的格子能访问的格子。程序应移到下次移动时访问性值最小的格子。

4.25（骑士旅行：强制方法）练习4.24开发了骑士旅行问题的解决方法，这个方法用“访问性试探”能产生许多解法并可以有效地执行。

随着计算机的运算能力越来越强，可以用更简单的算法解决这个问题，即采用强制算法。

- a) 用随机数产生程序让骑士在棋盘上按L形走法任意移动。程序一步一步走完这个棋盘。骑士能走多远？

- b) 通常，上述程序不会走太远。现在修改程序，进行1000次走法。用单下标数组跟踪每次走了几步。程序完成1000次旅行后，以表格形式打印这些信息。最佳结果是什么？

- c) 通常，上述程序能得到较好地走法，但无法走遍棋盘。现在删除次数限制，让程序一直运行，直到找出走遍棋盘的方法（注意，可能要好几个小时才能在强大的计算机上完成）。以表格形式打印这些信息，看看要多少次才能找出走遍棋盘的方法，花多少时间。

- d) 比较强制算法与前面介绍的访问性试探方法。哪个需要更认真地分析问题，哪个算法更难开发，哪个需要更强大的计算机功能，利用访问性试探能否事先保证找出走遍棋盘的方法，利用强制算法能否事先保证找出走遍棋盘的方法，比较两种方法的利与弊。

4.26（八皇后）国际象棋中的另一个难题是八皇后问题。简单地说，空棋盘上能否放八个皇后，使一个皇后不会“攻击”另一个皇后，即不会有两个皇后在同一行、同一列或同一对角线上。用练习4.24的思路设计解决八皇后问题的算法（提示：棋盘上的每一格可以指定一个值，表示一个皇后可以“删除”多少个格子，每个角指定数值22，如图4.26）。在64个格子中放上这些“删除”数之后，问题就变成：将下一个皇后放在删除数最少的格子中。

```

*****
**
* *
* *
* *
* *
* *
* *

```

图4.26 左上角的“删除”数为22

4.27 (八皇后: 强制算法) 这个练习要用几个强制算法解决练习 4.26 介绍的八皇后问题。

- a) 用练习 4.25 介绍的随机强制算法解决八皇后问题。
- b) 用穷举法, 即测试八个皇后在棋盘上的各种组合。
- c) 说明穷举法为什么不适用于骑士旅行问题?
- d) 比较随机强制算法与穷举法。

4.28 (骑士旅行: 闭合线路) 在骑士旅行问题中, 骑士经过 64 格中的每一格一次, 且只经过一次。闭合线路就是最后又回到出发的地方。修改练习 4.24 的骑士旅行程序, 测试闭合线路是否走遍了棋盘。

4.29 (Eratosthenes 筛选法) 质数是只能被 1 和本身整除的数。Eratosthenes 筛选法是一种寻找质数的方法, 这个方法如下所示:

- a) 生成一个数组, 将所有元素初始化为 1 (真)。下标为质数的数组元素保持 1, 所有其他数组元素最终设置为 0。
- b) 从数组下标 2 开始 (下标 1 为质数), 每次找到数值为 1 的数组元素时, 对数组余下部分循环, 将下标为该下标倍数的元素设置为 0。对于数组下标 2, 数组中下标为 2 的倍数的所有元素 (除 2 本身, 如 4、6、8、10 等等) 都设置为 0; 对于数组下标 3, 数组中下标为 3 的倍数的所有元素 (除 3 本身, 如 3、6、9、12、15 等等) 都设置为 0, 依次类推。

这个过程完成之后, 如果数组元素还是 1, 则下标为质数。这些下标可以打印出来。编写一个程序, 用 1000 个元素的数组确定和打印 1 到 999 之间的素数, 忽略数组中的元素 0。

4.30 (桶排序) 桶排序从要排序的单个下标正整数的数组开始, 并有一个双下标整数数组, 行下标为 0 到 9, 列下标为 0 到  $n-1$ , 其中  $n$  为要排序数组中的数值个数。每一行双下标整数数组称为一个桶。编写一个 bucketSort 函数, 取整数数组和数组长度参数, 并进行下列操作:

- a) 将单下标数组的每个值放到桶数组的行中 (根据该值的个位)。例如, 97 放在第 7 行, 3 放在第 3 行, 100 放在第 0 行, 称为“分布传递”。
- b) 一行一行地对桶数组进行循环, 将值复制回原数组, 称为“收集传递”。上述值在单下标数组中的新顺序为 100、3、97。
- c) 对十位、百位、千位等重复上述过程。

第二遍, 100 放在第 0 行, 3 放在第 0 行 (因为 3 没有十位), 97 放在第 9 行。经过收集传递之后, 上述值在单下标数组中的新顺序为 100、3、97。第三遍, 将 100 放在第 1 行, 3 和 97 放在第 0 行。经过收集传递之后, 上述值在单下标数组中的新顺序为所要的排序结果。

注意, 双下标桶数组的长度是要排序的整数数组长度的 10 倍。这种排序方法提供比冒泡排序更好的性能, 但要求更多的内存。冒泡排序方法只要多一个数据元素的空间。这就是用空间交换时间的范例: 桶排序方法比冒泡排序更快, 但使用更多内存。这个存储桶排序方法要求每遍都将所有数据复制回原数组中。另一种方法是生成第二个双下标数组, 在两个桶数组之间重复交换数据。

## 递归练习

4.31 (选择排序) 选择排序查找数组中的最小元素。然后将最小元素与数组中第一个元素交换。从第二个数组元素开始的子数组重复这个过程。每一次都把一个元素放到正确的位置。



置。这种排序与冒泡排序相似,对于  $n$  个元素的数组,要  $n-1$  遍,对每个子数组,要用  $n-1$  次比较以求得最小值。处理包含一个元素的子数组时,数组已经排序完毕。编写递归函数 `selectionSort`,完成这个算法。

- 4.32 (回文) 回文就是正读反读都一样的字符串,例如 “radar”, “able was i ere i saw elba” 和 “a man a plan a canal panama” (如果忽略空格)。编写递归函数 `testPalindrome`,在数组中的字符串为回文时返回 `true`,否则返回 `false`。函数忽略字符串中的空格和标点符号。
- 4.33 (线性查找) 修改图 4.19,用递归函数 `linearSearch` 对数组进行线性查找,函数应收到整型数组和数组长度参数。如果找到查找键,则返回数组下标,否则返回 `-1`。
- 4.34 (折半查找) 修改图 4.20,用递归函数 `binarySearch` 对数组进行折半查找。函数应收到整型数组和开始下标与结束下标参数。如果找到查找键,则返回数组下标,否则返回 `-1`。
- 4.35 (八皇后) 修改练习 4.26 的八皇后程序,用递归方法解决问题。
- 4.36 (打印数组) 编写递归函数 `printArray`,取数组和长度参数,不返回任何内容。函数在收到长度为 0 的数组时停止处理并返回。
- 4.37 (逆向打印字符串) 编写函数 `stringReverse`,取包含字符串的字符数组参数,逆向打印字符串且不返回任何内容。函数在收到 `null` 终止符时停止处理并返回。
- 4.38 (寻找数组中的最小值) 编写递归程序 `recursiveMinimum`,取数组和长度参数并返回数组中的最小元素,当函数在收到长度为 1 的数组时停止处理并返回。

## 第5章 指针与字符串

### 教学目标

- 能够使用指针
- 能用指针按引用调用向函数传递参数
- 了解指针、数组与字符串之间的密切关系
- 了解指针在函数中的使用
- 能够声明和使用字符串数组

### 5.1 简介

本章介绍 C++ 编程语言一个最强大的特性——指针。指针是 C++ 中最难掌握的问题之一。第3章介绍了引用可以用于实现按引用调用。指针使程序可模拟按引用调用，生成与操作动态数据结构，即能够伸缩的数据结构，如链表、队列、堆栈和树。本章介绍基本的指针概念，而且强调了数组、指针与字符串之间的密切关系，并包括一组很好的字符串操作练习。

第6章介绍结构中的指针使用。第9章和第10章介绍如何用指针和引用进行面向对象编程。第15章介绍动态内存管理技术以及生成和使用动态数据结构的例子。

把数组和字符串看成指针是从C语言演变而来的。本书后面会介绍把数组和字符串当作成熟的对象。

### 5.2 指针变量的声明与初始化

指针变量的值为内存地址。通常变量直接包含特定值，而指针则包含特定值变量的地址。因此可以说，变量名直接（directly）引用数值，而指针间接（indirectly）引用数值（如图5.1）。通过指针引用数值称为间接引用。

指针和任何其他变量一样，应先声明后使用。下列声明：

```
int *countPtr, count;
```

声明变量countPtr的类型为int\*（即指向整型值的指针），或者说成“countPtr是int的指针”或“countPtr指向整数类型的对象”。变量count声明为整数，而不是整型值的指针。声明中的\*只适用于countPtr。声明为指针的每个变量前面都要加上星号（\*）。例如，下列声明：

```
float *xPtr, *yPtr;
```

表示xPtr和yPtr都是指向float值的指针。声明中以这种方式使用\*时，它表示变量声明为指针。指针可以声明为指向任何数据类型的对象。

**常见编程错误 5.1**

假设\* 对指针的声明会分配到声明中逗号分隔的指针变量名列表中的所有指针变量名,从而将指针声明为非指针。声明为指针的每个变量前面都要加上星号(\*)。

**编程技巧 5.1**

尽管不是必需的,但在指针变量名中加上 ptr 字样能更清楚地表示这些变量是指针,需要相应的处理。

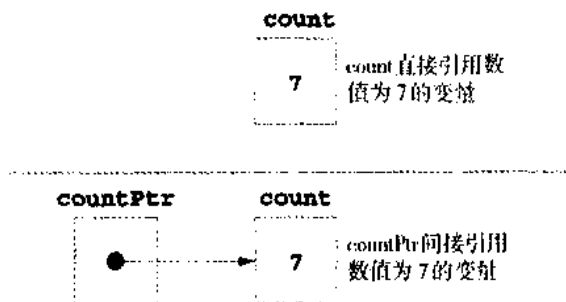


图 5.1 直接和间接引用变量

指针应在声明时或在赋值语句中初始化。指针可以初始化为0、NULL或一个地址。数值为0或NULL的指针不指任何内容。NULL是头文件<iostream.h> (和另外几个标准库头文件)中定义的符号化常量。将指针初始化为NULL等于将指针初始化为0,但C++中优先选择0。指定0时,它变为指针的相应类型。数值0是惟一可以不将整数转换为指针类型而直接赋给指针变量的整数值。5.3节将介绍将变量地址赋给指针。

**测试与调试提示 5.1**

初始化指针以防止其指向未知的或未初始化的内存区。

## 5.3 指针运算符

& (地址)运算符是个一元运算符,返回操作数的地址。例如,假设声明:

```
int y = 5;
int *yPtr;
```

则下列语句:

```
yPtr = &y;
```

将变量y的地址赋给指针变量yPtr。变量yPtr“指向”y。图5.2显示了执行上述语句之后的内存示意图。图中从指针向所指对象画一个箭头,表示“指向关系”。

图5.3显示了指针在内存中的表示,假设整型变量y存放在地址600000,指针变量yPtr存放在地址500000。地址运算符的操作数应为左值,(即要赋值的项目,如变量名),地址运算符不能用于常量、不产生引用的表达式和用存储类register声明的变量。

“\*”运算符通常称为间接运算符(indirection operator)或复引用运算符(dereferencing operator),返回操作数(即指针)所指对象的同义词、别名或浑名。例如(图5.2再次引用),下列语句:

```
cout << * yPtr << endl;
```

指向变量 *y* 的值 (5)，如同下列语句：

```
cout << y << endl;
```

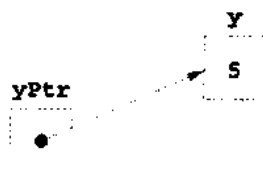


图 5.2 指针指向内存中整数变量的示意图

这里使用 *\** 的方法称为复引用指针 (dereferencing a pointer)。注意复引用指针也可以用于赋值语句左边，例如下列语句：

```
*yPtr = 9;
```

将数值 9 赋给图 5.3 中的 *y*。复引用指针也可用于接收输入值，例如：

```
cin >> * yPtr;
```

复引用的指针是个左值。



图 5.3 指针在内存中的表示

#### 常见编程错误 5.2

如果指针没有正确地初始化或没有指定指向内存中的特定地址，则复引用指针可能造成致命的运行时错误，或者意外修改重要数据。虽然运行完程序，但得到的是错误结果。

#### 常见编程错误 5.3

复引用非指针是个语法错误。

#### 常见编程错误 5.4

复引用 0 指针通常是个致命的运行时错误。

图 5.4 的程序演示了指针运算符。本例中通过 *<<* 用十六进制整数输出内存地址 (十六进制整数见附录“数值系统”)。

#### 可移植性提示 5.1

输出指针的格式与机器有关，有些系统用十六进制整数，而有些系统用十进制整数。

注意 *a* 的地址和 *aPtr* 的值在输出中是一致的，说明 *a* 的地址实际赋给了指针变量 *aPtr*。& 和 \* 运算符是互逆的，如果两者同时作用于 *aPtr*，则打印相同的结果。图 5.5 显示了前面所介绍的运算符的优先级和结合律。

```

1 // Fig. 5.4: fig05_04.cpp
2 // Using the & and * operators
3 #include <iostream.h>
4
5 int main()
6 {
7     int a;          // a is an integer
8     int *aPtr;      // aPtr is a pointer to an integer
9
10    a = 7;
11    aPtr = &a;       // aPtr set to address of a
12
13    cout << "The address of a is " << &a
14          << "\nThe value of aPtr is " << aPtr;
15
16    cout << "\n\nThe value of a is " << a
17          << "\nThe value of *aPtr is " << *aPtr;
18
19    cout << "\n\nShowing that * and & are inverses of "
20          << "each other.\n&*aPtr = " << &*aPtr
21          << "\n*&aPtr = " << *&aPtr << endl;
22    return 0;
23 }

```

**输出结果：**

```

The address of a is 0x0064FDF4
The value of aPtr is 0x0064FDF4
The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
&*aPtr = 0x0064FDF4
*&aPtr = 0x0064FDF4

```

图 5.4 &amp; 与 \* 指针运算符

运算符	结合律	类型
() []	从左向右	括号
++ -- + - static_cast<type>()	从右向左	一元
& *		
* / %	从左向右	乘
+ -	从左向右	加
<< >>	从左向右	插入/读取
< <= > >=	从左向右	关系
== !=	从左向右	相等
&&	从左向右	逻辑 AND
	从左向右	逻辑 OR
?:	从右向左	条件
= += -= *= /= %=	从右向左	赋值
,	从左向右	逗号

图 5.5 运算符的优先级和结合律

## 5.4 按引用调用函数

C++ 用三种方式向函数传递数值：按值调用（call-by-value）、用引用参数按引用调用（call-by-reference reference argument）和用指针参数按引用调用（call-by-reference pointer argument）。第3章比较了按引用调用与按值调用，本章主要介绍用指针参数按引用调用。

第3章曾介绍过，return 可以从被调用函数向调用者返回一个值（或不返回值而从被调用函数返回控制）。我们还介绍了用引用参数将参数传递给函数，使函数可以修改参数的原有值（这样可以从函数“返回”多个值），或将大的数据对象传递给函数而避免按值调用传递对象的开销（即复制对象所需的开销）。指针和引用一样，也可以修改调用者的一个或几个变量，或将大的数据对象指针传递给函数而避免按值调用传递对象的开销。

在C++中，程序员可以用指针和间接运算符模拟按引用调用（就像C语言程序中的按引用调用一样）。调用函数并要修改参数时，传递该参数地址，通常要在要修改数值的变量名前面加上地址运算符（&）。第4章曾介绍过，数组不能用地址运算符（&）传递，因为数组名是内存中数组的开始位置（数组名等同于&arrayName[0]），即数组名已经是个指针。向函数传递参数地址时，可以在函数中使用间接运算符形成变量名的同义词、别名或浑名，并可用其修改调用者内存中该地址的值（如果变量不用const声明）。

图5.6和5.7的程序是计算整数立方函数的两个版本cubeByValue和cubeByReference。图5.6按值调用将变量number传递给函数cubeByValue。函数cubeByValue求出参数的立方，并将新值用return语句返回main，并在main中将新值赋给number。可以先检查函数调用的结果再修改变量值。例如，在这个程序中，可以将cubeByValue的结果存放在另一变量中，检查其数值，然后再将新值赋给number。

```
1 // Fig. 5.6: fig05_06.cpp
2 // Cube a variable using call-by-value
3 #include <iostream.h>
4
5 int cubeByValue( int );    // prototype
6
7 int main()
8 {
9     int number = 5;
10
11     cout << "The original value of number is " << number;
12     number = cubeByValue( number );
13     cout << "\nThe new value of number is " << number << endl;
14     return 0;
15 }
16
17 int cubeByValue( int n )
18 {
19     return n * n * n;    // cube local variable n
20 }
```

**输出结果：**

```
The original value of number is 5
The new value of number is 125
```

图 5.6 按值调用求出参数的立方

图 5.7 的程序按引用调用传递变量 `number`（传递 `number` 的地址）到函数 `cubeByReference`。函数 `cubeByReference` 取 `nPtr`（`int` 的指针）作为参数。函数复引用指针并求出 `nPtr` 所指值的立方，从而改变 `main` 中的 `number` 值。图 5.8 和 5.9 分别分析了图 5.6 和 5.7 所示程序。

```
1 // Fig. 5.7: fig05_07.cpp
2 // Cube a variable using call-by-reference
3 // with a pointer argument
4 #include <iostream.h>
5
6 void cubeByReference( int * ); // prototype
7
8 int main()
9 {
10     int number = 5;
11
12     cout << "The original value of number is " << number;
13     cubeByReference( &number );
14     cout << "\nThe new value of number is " << number << endl;
15     return 0;
16 }
17
18 void cubeByReference( int *nPtr )
19 {
20     *nPtr = *nPtr * *nPtr * *nPtr; // cube number in main
21 }
```

**输出结果：**

```
The original value of number is 5
The new value of number is 125
```

图 5.7 用指针参数按引用调用求出参数的立方

**常见编程错误 5.5**

要复引用指针以取得指针所指的值时不复引用指针是个错误。

接收地址参数的函数要定义接收地址的指针参数。例如，`cubeByReference` 的函数首部如下所示：

```
void cubeByReference( int *nPtr )
```

这个函数首部指定函数 `cubeByReference` 接收整型变量的地址（即整型指针）作为参数，在 `nPtr` 中局部存放地址，不返回值。

`cubeByReference` 的函数原型包含括号中的 `int *`。和其他变量类型一样，不需要在函数原型中包括指针名。参数名仅用于程序中的说明，编译器将其忽略。

在需要单下标数组参数的函数首部和函数原型中，可以用 `cubeByReference` 参数表中的指针符号。编译器并不区分接收指针的函数和接收单下标数组的函数。当然，函数必须“知道”何时接收数组或要进行按引用调用的单个变量。编译器遇到形如 `int b[]` 的单下标数组函数参数时，编译器将参数变为指针符号 `int * const b`（`b` 是指向整数的常量指针），`const` 见第 5.5 节介绍。声明函数参数为单下标数组的两种形式可以互换。

## 编程技巧 5.2

除非调用者显式要求被调用函数修改调用者环境中参数变量的值，否则按值调用将参数传递给函数。这是最低权限原则的另一个例子。

main 调用 cubeByValue 之前:

<pre>int main() {     int number = 5;      number = cubeByValue( number ); }</pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">number</div> 5	<pre>int cubeByValue( int n ) {     return n * n * n; }</pre> <div style="text-align: right;">n 未定义</div>
--	--	---

cubeByValue 接收调用之后:

<pre>int main() {     int number = 5;      number = cubeByValue( number ); }</pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">number</div> 5	<pre>int cubeByValue( int n ) {     return n * n * n; }</pre> <div style="text-align: right;">n 5</div>
--	--	---

cubeByValue 计算参数 n 的立方值之后:

<pre>int main() {     int number = 5;      number = cubeByValue( number ); }</pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">number</div> 5	<pre>int cubeByValue( int n ) {     return 125 * n; }</pre> <div style="text-align: right;">n 未定义</div>
--	--	---

cubeByValue 返回 main 之后:

<pre>int main() {     int number = 5;      number = 125 * cubeByValue( number ); }</pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">number</div> 5	<pre>int cubeByValue( int n ) {     return n * n * n; }</pre> <div style="text-align: right;">n 未定义</div>
--	--	---

main 完成对 number 赋值之后:

<pre>int main() {     int number = 5;      number = cubeByValue( number ); }</pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">number</div> 125	<pre>int cubeByValue( int n ) {     return n * n * n; }</pre> <div style="text-align: right;">n 未定义</div>
--	--	---

图 5.8 典型的按值调用分析



按引用调用函数 `cubeByReference` 之前:

<code>int main()</code>	<code>number</code>	<code>void cubeByReference( int *nPtr )</code>
{	5	{
<code>int number = 5;</code>		<code>*nPtr = *nPtr * *nPtr * *nPtr;</code>
<code>cubeByReference( &amp;number );</code>		}
}		<code>nPtr</code>
		未定义

按引用调用函数 `cubeByReference` 之后和计算 `*nPtr` 立方值之前:

<code>int main()</code>	<code>number</code>	<code>void cubeByReference( int *nPtr )</code>
{	5	{
<code>int number = 5;</code>		<code>*nPtr = *nPtr * *nPtr * *nPtr;</code>
<code>cubeByReference( &amp;number );</code>		}
}		<code>nPtr</code>
		•

计算 `*nPtr` 立方值之后:

<code>int main()</code>	<code>number</code>	<code>void cubeByReference( int *nPtr )</code>
{	125	{
<code>int number = 5;</code>		<code>*nPtr = *nPtr * *nPtr * *nPtr;</code>
<code>cubeByReference( &amp;number );</code>		}
}		<code>nPtr</code>
		•

图 5.9 典型的用指针参数按引用调用分析

## 5.5 指针与常量限定符

`const` 限定符可以使程序员告诉编译器特定变量的值不能修改。

### 软件工程视点 5.1

`const` 限定符可以执行最低权限原则。利用最低权限原则正确设计软件可以大大减少调试时间和不正确的副作用,使程序更容易修改与维护。

### 可移植性提示 5.2

尽管 ANSI C 和 C++ 中定义了 `const` 限定符,但有些编译器无法正确实现。

几年来,大量 C 语言遗留代码都是在没有 `const` 限定符的情况下编写的。因此,使用旧版 C 语言代码的软件工程有很大的改进空间。许多目前使用 ANSI C 和 C++ 的程序员也没有在程序中使用 `const` 限定符,因为他们是从 C 语言的早期版本开始编程的,这些程序员错过了许多改进软件工程的好机会。

函数参数使用或不用 `const` 限定符的可能性有六种,两种用按值调用传递参数,四种按引用调用传递参数,根据最低权限原则来进行选择。在参数中向函数提供完成指定任务所需的数据访问,但不要提供更多权限。

第3章曾经介绍,按值调用传递参数时,函数调用中要生成参数副本并将其传递给函数。如果函数中修改副本,则调用者的原值保持不变。许多情况下,需要修改传入函数的值以使函数能够完成任务。但有时即使被调用函数只是操作原值的副本,也不能在被调用函数中修改这个值。

假设函数取一个单下标数组及其长度为参数,并打印数值。这种函数应对数组进行循环并分别输出每个数组元素。函数体中用数组长度确定数组的最高下标,以便在打印完成后结束循环。在函数体中不能改变这个数组长度。

#### 软件工程视点 5.2

如果函数体中不能修改传递的值,则这个参数应声明为 `const` 以避免被意外修改。

如果试图修改 `const` 类型的值,则编译器会捕获这个错误并发出一个警告或错误消息(取决于特定的编译器)。

#### 软件工程视点 5.3

按值调用时,只能在调用函数中改变一个值。这个值通过函数返回值进行赋值。要在调用函数中改变多个值,就要按引用传递多个参数。

#### 编程技巧 5.3

使用函数之前,检查函数原型以确定可以修改的参数。

将指针传递给函数有四种方法:非常量数据的非常量指针、常量数据的非常量指针、非常量数据的常量指针和常量数据的常量指针。每种组合提供不同的访问权限。

最高访问权限是非常量数据的非常量指针,可以通过复引用指针而修改,指针可以修改成指向其他数据。声明非常量数据的非常量指针时不用 `const`。这种指针可以接收函数中的字符串,用指针算法处理或修改字符串中的每个字符。图 5.10 中的函数 `convertToUppercase` 声明参数 `sPtr (char * sPtr)` 为非常量数据的非常量指针。函数用指针算法一次一个字符地处理字符串 `string`。字符串中 'a' 到 'z' 的字符用函数 `toupper` 变为相应的大写字母,其余字符不变。函数 `toupper` 取一个字符作为参数。如果是小写字母,则返回相应的大写字母,否则返回原字符。函数 `toupper` 是字符处理库 `ctype.h` 中(见第 16 章)的一部分。

常量数据的非常量指针,指针可以修改成指向其他数据,但数据不能通过指针修改。这种指针可以接收函数的数组参数,函数处理数组每个元素而不修改数据。例如,图 5.11 的函数 `printCharacters` 将参数 `sPtr` 声明为 `const char *` 类型,表示“`sPtr` 是字符常量的指针”。函数体用 `for` 循环输出字符串中的每个字符,直到遇到 `null` 终止符。打印每个字符之后,指针 `sPtr` 递增,指向字符串中下一个字符。

```
1 // Fig. 5.10: fig05_10.cpp
2 // Converting lowercase letters to uppercase letters
3 // using a non-constant pointer to non-constant data
4 #include <iostream.h>
5 #include <ctype.h>
6
7 void convertToUppercase( char * );
8
9 int main()
10 {
11     char string[] = "characters and $32.98";
12
13     cout << "The string before conversion is: " << string;
14     convertToUppercase( string );
15     cout << "\nThe string after conversion is: "
16         << string << endl;
```

```

17     return 0;
18 }
19
20 void convertToUppercase( char *sPtr )
21 {
22     while ( *sPtr != '\0' ) {
23
24         if ( *sPtr >= 'a' && *sPtr <= 'z' )
25             *sPtr = toupper( *sPtr ); // convert to uppercase
26
27         ++sPtr; // move sPtr to the next character
28     }
29 }

```

**输出结果:**

The string before conversion is: characters and \$32.98  
The string after conversion is: CHARACTERS AND \$32.98

图5.10 将字符串变成大写

```

1 // Fig. 5.11: fig05_11.cpp
2 // Printing a string one character at a time using
3 // a non-constant pointer to constant data
4 #include <iostream.h>
5
6 void printCharacters( const char * );
7
8 int main()
9 {
10     char string[] = "print characters of a string";
11
12     cout << "The string is:\n";
13     printCharacters( string );
14     cout << endl;
15     return 0;
16 }
17
18 // In printCharacters, sPtr is a pointer to a character
19 // constant. Characters cannot be modified through sPtr
20 // (i.e., sPtr is a "read-only" pointer).
21 void printCharacters( const char *sPtr )
22 {
23     for ( ; *sPtr != '\0'; sPtr++ ) // no initialization
24         cout << *sPtr;
25 }

```

**输出结果:**

The string is:  
print characters of a string

图5.11 用常量数据的非常量指针打印字符串（一次打印一个字符）

图5.12演示了函数接收常量数据的非常量指针，并试图通过指针修改数据在编译时产生的语法错误消息。

```
1 // Fig. 5.12: fig05_12.cpp
2 // Attempting to modify data through a
3 // non-constant pointer to constant data.
4 #include <iostream.h>
5
6 void f( const int * );
7
8 int main()
9 {
10     int y;
11
12     f( &y );      // f attempts illegal modification
13
14     return 0;
15 }
16
17 // In f, xPtr is a pointer to an integer constant
18 void f( const int *xPtr )
19 {
20     *xPtr = 100;  // cannot modify a const object
21 }
```

**输出结果:**

Compiling FIG05\_12.CPP:

Error FIG05\_12.CPP 20: Cannot modify a const object

Warning FIG05\_12.CPP 21: Parameter 'xPtr' is never used

图 5.12 试图通过常量数据的非常量指针修改数据

众所周知,数组是累计数据类型,用同一名称存放相同类型的相关数据项。第6章将介绍另一种形式的累计数据类型——结构(structure),也称为记录(record)。结构可以用同一名称存放不同类型的相关数据项(例如,存放公司每个员工的信息)。调用带数组参数的函数时,数组模拟按引用调用自动传递给函数。但结构则总是按值调用,传递整个结构的副本。这就需要复制结构中每个数据项目并将其存放在计算机函数调用堆栈中的执行时开销(函数执行时用函数调用堆栈存放函数调用中使用的局部变量)。结构数据要传递给函数时,可以用常量数据的指针(或常量数据的引用)得到按引用调用的性能和按值调用对数据的保护。传递结构的指针时,只要复制存放结构的地址。在4字节地址的机器上,只要复制4字节内存而不是复制结构的几百或几千字节。

**性能提示 5.1**

要传递结构之类的大对象时,可以用常量数据的指针(或常量数据的引用)得到按引用调用的性能和按值调用对数据的保护。

非常量数据的常量指针总是指向相同的内存地址,该地址中的数据可以通过指针修改。这里的数组名是默认的。数组名是数组开头的常量指针,数组中的所有数据可以用数组名和数组下标访问和修改。非常量数据的常量指针可以接收数组为函数参数,该函数只用数组下标符号访问数组元素。声明为const的指针应在声明时初始化(如果是函数参数,则用传入函数的指针初始化)。图5.13的程序想修改常量指针,指针ptr的类型声明为int\* const,图中的声明表示“ptr是整数的常量指针”,指针用整型变量x的地址初始化。程序要将y的地址赋给ptr,但产生一个错误消息。注意数值7赋给\* ptr时不产生错误,说明ptr所指的值是可修改的。

**常见编程错误 5.6**

声明为 const 的指针不在声明时初始化是个语法错误。

常量数据的常量指针的访问权限最低。这种指针总是指向相同的内存地址,该内存地址的数据不能修改。数组传递到函数中,该函数只用数组下标符号读取,而不能修改数组。图5.14的程序演示声明指针变量 ptr 为 const int \* const, 表示“ptr 是常量整数的常量指针”。图中显示了修改 ptr 所指数据和修改存放指针变量的地址时产生的错误消息。注意输出 ptr 所指的值时不产生错误,因为输出语句中没有进行修改。

```
1 // Fig. 5.13: fig05_13.cpp
2 // Attempting to modify a constant pointer to
3 // non-constant data
4 #include <iostream.h>
5
6 int main()
7 {
8     int x, y;
9
10    int * const ptr = &x; // ptr is a constant pointer to an
11                          // integer. An integer can be modified
12                          // through ptr, but ptr always points
13                          // to the same memory location.
14    *ptr = 7;
15    ptr = &y;
16
17    return 0;
18 }
```

**输出结果:**

Compiling FIG05\_13.CPP:

Error FIG05\_13.CPP 15: Cannot modify a const object

Warning FIG05\_13.CPP 18: 'y' is declared but never used

图 5.13 修改非常量数据的常量指针

```
1 // Fig. 5.14: fig05_14.cpp
2 // Attempting to modify a constant pointer to
3 // constant data.
4 #include <iostream.h>
5
6 int main()
7 {
8     int x = 5, y;
9
10    const int *const ptr = &x; // ptr is a constant pointer to a
11                              // constant integer. ptr always
12                              // points to the same location
13                              // and the integer at that
14                              // location cannot be modified.
15    cout << *ptr << endl;
16    *ptr = 7;
17    ptr = &y;
18
19    return 0;
20 }
```

```
20 }
```

**输出结果:**

```
Compiling FIG05_14.CPP:
Error FIG05_14.CPP 16: Cannot modify a const object
Error FIG05_14.CPP 17: Cannot modify a const object
Warning FIG05_14.CPP 20: 'y' is declared but never used
```

图 5.14 修改常量数据的常量指针

## 5.6 按引用调用的冒泡排序

下面将图4.16的冒泡排序程序修改成用两个函数bubbleSort和swap(如图5.15)。函数bubbleSort进行数组排序,它调用函数swap,变换数组元素array[j]和array[j+1]。记住,C++强制函数之间的信息隐藏,因此swap并不能访问bubbleSort中的各个元素。由于bubbleSort要求swap访问交换的数组元素,因此bubbleSort要将这些元素按引用调用传递给swap,每个数组元素的地址显式传递。尽管整个数组自动按引用调用传递,但各个数组元素是标量,通常按值调用传递。因此,bubbleSort对swap调用中的每个数组元素使用地址运算符(&),如下所示的语句:

```
swap( &array[ j ], &array[ j + 1 ] );
```

实现按引用调用。函数swap用指针变量element1Ptr接收&array[j]。由于信息隐藏,swap并不知道名称&array[j],但swap可以用\*element1Ptr作为&array[j]的同义词。这样,swap引用\*element1Ptr时,实际上是引用bubbleSort中的&array[j]。同样,swap引用\*element2Ptr时,实际上是引用bubbleSort中的array[j+1]。虽然swap不能用:

```
hold = array[ j ];
array[ j ] = array[ j + 1 ];
array[ j + 1 ] = hold;
```

但图5.15中的swap函数用:

```
hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

达到相同的效果。

```
1 // Fig. 5.15: fig05_15.cpp
2 // This program puts values into an array, sorts the values into
3 // ascending order, and prints the resulting array.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 void bubbleSort( int *, const int );
8
9 int main()
10 {
11     const int arraySize = 10;
12     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13     int i;
14
```

```

15  cout << "Data items in original order\n";
16
17  for ( i = 0; i < arraySize; i++ )
18      cout << setw( 4 ) << a[ i ];
19
20  bubbleSort( a, arraySize );           // sort the array
21  cout << "\nData items in ascending order\n";
22
23  for ( i = 0; i < arraySize; i++ )
24      cout << setw( 4 ) << a[ i ];
25
26  cout << endl;
27  return 0;
28 }
29
30 void bubbleSort( int *array, const int size )
31 {
32     void swap( int *, int * );
33
34     for ( int pass = 0; pass < size - 1; pass++ )
35
36         for ( int j = 0; j < size - 1; j++ )
37
38             if ( array[ j ] > array[ j + 1 ] )
39                 swap( &array[ j ], &array[ j + 1 ] );
40 }
41
42 void swap( int *element1Ptr, int *element2Ptr )
43 {
44     int hold = *element1Ptr;
45     *element1Ptr = *element2Ptr;
46     *element2Ptr = hold;
47 }

```

**输出结果：**

```

Data items in original order
 2   6   4   8   10  12  89   68   45   37
Data items in ascending order
 2   4   6   8   10  12  37   45   68   89

```

图 5.15 按引用调用的冒泡排序

注意函数 bubbleSort 中的几个特性。函数首部中将 array 声明为 int \* array 而不是 int array[], 表示 bubbleSort 接收单下标数组作为参数 ( 这些符号是可以互换的 )。参数 size 声明为 const 以保证最低权限原则。尽管参数 size 接收 main 中数值的副本, 且修改该副本并不改变 main 中的值, 但是 bubbleSort 不必改变 size 即可完成任务。bubbleSort 执行期间数组的长度保持不变, 因此, size 声明为 const 以保证不被修改。如果排序过程中修改数组长度, 则排序算法无法正确执行。

bubbleSort 函数体中包括了函数 swap 的原型, 因为它是调用 swap 的惟一函数。将原型放在 bubbleSort 中, 使得只能从 bubbleSort 正确地调用 swap。其他函数要调用 swap 时无法访问正确的函数原型, 这通常会造成语法错误, 因为 C++ 需要函数原型。

**软件工程视点 5.4**

将函数原型放在其他函数中能保证最低权限原则, 只能从该原型所在函数中正确地调用。

注意函数 bubbleSort 接收数组长度参数。函数必须知道数组长度才能排序数组。数组传递到函数时，函数接收数组第一个元素的内存地址。数组长度要单独传递给函数。

通过将函数 bubbleSort 定义成接收数组长度作为参数，可以让函数在排序任何长度单下标整型数组的程序中使用。

#### 软件工程视点 5.5

向函数传递数组时，同时传递数组长度（而不是在函数中建立数组长度信息），这样能使函数更加一般化，以便在许多程序中复用。

数组长度可以直接编程到函数内，这样会把函数的使用限制在特定长度的数组并减少其复用性。程序中只有处理特定长度的单下标整型数组时才能使用这个函数。

C++ 提供一元运算符 sizeof，确定程序执行期间的数组长度或其他数据类型长度（字节数）。采用数组名时（如图 5.16 所示），sizeof 运算符返回数组中的总字节数为 size\_t 类型的值，通常是 unsigned int 类型。这里使用的计算机将 float 类型的变量存放在 4 字节内存中，array 声明为 20 个元素，因此 array 使用 80 字节内存空间。在接收数组参数的函数中采用指针参数时，sizeof 运算符返回指针长度的字节数（4）而不是数组长度。

#### 常见编程错误 5.7

在函数中用 sizeof 运算符寻找数组参数长度的字节数时返回指针长度的字节数而不是数组长度的字节数。

```
1 // Fig. 5.16: fig05_16.cpp
2 // Sizeof operator when used on an array name
3 // returns the number of bytes in the array.
4 #include <iostream.h>
5
6 size_t getSize( float * );
7
8 int main()
9 {
10     float array[ 20 ];
11
12     cout << "The number of bytes in the array is "
13          << sizeof( array )
14          << "\nThe number of bytes returned by getSize is "
15          << getSize( array ) << endl;
16
17     return 0;
18 }
19
20 size_t getSize( float *ptr )
21 {
22     return sizeof( ptr );
23 }
```

#### 输出结果：

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

图 5.16 采用数组名时，sizeof 运算符返回数组中的总字节数



数组中的元素个数也可以用两个 `sizeof` 操作的结果来确定。例如，考虑下列数组声明：

```
double realArray[ 22 ];
```

如果 `double` 数据类型的变量存放在 8 个字节的内存中，则数组 `realArray` 总共包含 176 个字节。要确定数组中的元素个数，可以用下列表达式：

```
sizeof realArray / sizeof( double )
```

这个表达式确定 `realArray` 数组中的字节数，并将这个值除以内存中存放一个 `double` 值的字节数。

图 5.17 的程序用 `sizeof` 运算符计算我们所用的个人计算机上存放每种标准数据类型时使用的字节数。

#### 可移植性提示 5.3

存放特定数据类型时使用的字节数随系统的不同而不同。编写的程序依赖于数据类型长度而且要在几个计算机系统中运行时，用 `sizeof` 确定存放这种数据类型时使用的字节数。

```
1 // Fig. 5.17: fig05_17.cpp
2 // Demonstrating the sizeof operator
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     char c;
9     short s;
10    int i;
11    long l;
12    float f;
13    double d;
14    long double ld;
15    int array[ 20 ], *ptr = array;
16
17    cout << "sizeof c = " << sizeof c
18         << "\tsizeof(char) = " << sizeof( char )
19         << "\nsizeof s = " << sizeof s
20         << "\tsizeof(short) = " << sizeof( short )
21         << "\nsizeof i = " << sizeof i
22         << "\tsizeof(int) = " << sizeof( int )
23         << "\nsizeof l = " << sizeof l
24         << "\tsizeof(long) = " << sizeof( long )
25         << "\nsizeof f = " << sizeof f
26         << "\tsizeof(float) = " << sizeof( float )
27         << "\nsizeof d = " << sizeof d
28         << "\tsizeof(double) = " << sizeof( double )
29         << "\nsizeof ld = " << sizeof ld
30         << "\tsizeof(long double) = " << sizeof( long double )
31         << "\nsizeof array = " << sizeof array
32         << "\nsizeof ptr = " << sizeof ptr
33         << endl;
34    return 0;
35 }
```

输出结果：

```
sizeof c = 1    sizeof(char) = 1
```

```

sizeof s = 2      sizeof(short) = 2
sizeof i = 4      sizeof(int) = 4
sizeof l = 4      sizeof(long) = 4
sizeof f = 4      sizeof(float) = 4
sizeof d = 8      sizeof(double) = 8
sizeof ld = 8     sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4

```

图 5.17 用 sizeof 运算符计算存放每种标准数据类型时使用的字节数

sizeof 运算符可以用于任何变量名、类型名或常量值。用于变量名（不是数组名）和常量值时，返回存放特定变量或常量类型所用的字节数。注意，如果提供类型名操作数，则 sizeof 使用的括号是必需的；如果提供变量名操作数，则 sizeof 使用的括号不是必需的。记住，sizeof 是个运算符而不是个函数。

#### 常见编程错误 5.8

如果提供类型名操作数，而不在 sizeof 操作中使用括号则是个语法错误。

#### 性能提示 5.2

sizeof 属于编译时的一元运算符，而不是个执行时函数。这样，使用 sizeof 并不对执行性能造成不良影响。

## 5.7 指针表达式与指针算法

指针是算术表达式、赋值表达式和比较表达式中的有效操作数。但是，通常并不是这些表达式中使用的所有运算符都在指针变量中有效。本节介绍可以用指针操作数的运算符及这些运算符的用法。

只有少量操作可以对指针进行。指针可以自增（++）或自减（--），整数可以加进指针中（+ 或 +=），也可以从指针中减去整数（- 或 -=），指针可以减去另一指针。

假设声明了数组 `int v[5]`，其第一个元素位于内存地址 3000。假设指针 `vPtr` 已经初始化为指向数组 `v[0]`，即 `vPtr` 的值为 3000。图 5.18 演示了在 32 位的机器中的这种情况。注意 `vPtr` 可以初始化为数组 `v` 的指针，如下所示：

```

vPtr = v;
vPtr = &v[ 0 ];

```

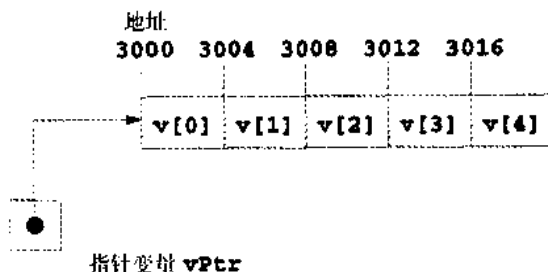


图 5.18 数组 `v` 和指向 `v` 的指针变量 `vPtr`

#### 可移植性提示 5.4

如今的大多数计算机都是 16 位或 32 位，有些较新的计算机用 8 字节整数。由于指针算法的结果取决于指针所指对象的长度，因此指针算法与机器有关。

按照传统算法,  $3000+2$  得到 3002。而指针算法通常不是这样。将指针增加或减去一个整数时, 指针并不是直接增加或减去这个整数, 而是加上指针所指对象长度的这个倍数。这些字节数取决于对象的数据类型。例如, 下列语句:

```
vPtr += 2;
```

在用 4 字节内存空间存储整数时得到的值为 3008 ( $3000 + 2 * 4$ )。对数组 `v`, 这时 `vPtr` 指向 `v[2]`, 如图 5.19。如果用 2 字节内存空间, 则上述结果得到 3004 ( $3000 + 2 * 2$ )。如果数组为不同数据类型, 则上述语句将指针递增指针所指对象长度的 2 倍。对字符数组进行指针算法时, 结果与普通算法相同, 因为每个字符的长度为一个字节。

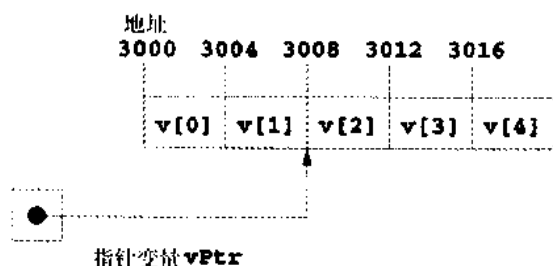


图 5.19 经过指针运算之后的 `vPtr`

如果 `vPtr` 递增到 3016, 指向 `v[4]`, 则下列语句:

```
vPtr -= 4;
```

将 `vPtr` 复位为 3000, 即数组开头。如果指针加 1 或减 1, 则可以用自增 (`++`) 和自减 (`--`) 运算符。下列语句:

```
++vPtr;  
vPtr++;
```

将指针移到数组中的下一个位置。下列语句:

```
--vPtr;  
vPtr--;
```

将指针移到数组中的前一个位置。

指针变量还可以相减。例如, 如果 `vPtr` 包含地址 3000, `v2ptr` 包含地址 3008, 则下列语句:

```
x = v2Ptr - vPtr;
```

将 `x` 指定为 `vPtr` 到 `v2Ptr` 的元素个数, 这里为 2。指针算法只在对数组进行时才有意义。我们不能假设两个相同类型的变量在内存中相邻的地址存放, 除非它们是数组的相邻元素。

#### 常见编程错误 5.9

对于不引用数组值的指针采用指针算法通常是个逻辑错误。

#### 常见编程错误 5.10

将两个不引用同一数组的指针相加或相减通常是个逻辑错误。

#### 常见编程错误 5.11

使用指针算法时超过数组边界通常是个逻辑错误。

如果两个指针的类型相同,则可以将一个指针赋给另一个指针。否则要用强制类型转换运算符将赋值语句右边的指针值转换为赋值语句左边的指针值。这个规则的例外是void的指针(即void\*),该指针是个一般性指针,可以表示任何指针类型。所有指针类型都可以赋给void指针而不需要类型转换。但是,void指针不能直接赋给另一类型的指针,而要先将void指针转换为正确的指针类型。

void\* 指针不能复引用。例如,编译器知道int指针指向32位机器中的4字节内存,但void指针只是包含未知数据类型的内存地址,指针所指的字节数是编译器所不知道的。编译器要知道数据类型才能确定该指针复引用时的字节数。对于void指针,无法从类型确定字节数。

#### 常见编程错误 5.12

除了void\* 类型外,将一种类型的指针赋给另一种类型的指针而不先将一种类型的指针转换为另一种类型的指针是个语法错误。

#### 常见编程错误 5.13

复引用 void\* 指针是个语法错误。

指针可以用相等和关系运算符比较,但这种比较只在对相同数组成员进行时才有意义。指针比较是对指针存放的地址进行比较。例如,比较指向同一数组的两个指针可以表示一个指针所指的元素号比另一个指针所指的元素号更高。指针比较常用于确定指针是否为0。

## 5.8 指针与数组的关系

C++中指针与数组关系密切,几乎可以互换使用。数组名可以看成常量指针,指针可以进行任何有关数组下标的操作。

#### 编程技巧 5.4

操作数组时用数组符号而不用指针符号。尽管程序编译时间可能稍长一些,但程序更加清晰。

假设声明了整数数组b[5]和整数指针变量bPtr。由于数组名(不带下标)是数组第一个元素的指针,因此可以用下列语句将bPtr设置为b数组第一个元素的地址:

```
bPtr = b;
```

这等于取数组第一个元素的地址,如下所示:

```
bPtr = &b[ 0 ];
```

数组元素b[3]也可以用指针表达式引用:

```
*( bPtr + 3 )
```

上述表达式中的3是指针的偏移量(offset)。指针指向数组开头时,偏移量表示要引用的数组元素,偏移量值等于数组下标。上述符号称为指针/偏移量符号(pointer/offset notation)。括号是必需的,因为\*的优先顺序高于+的优先顺序。如果没有括号,则上述表达式将表达式\*bPtr的值加上3(即3加到b[0]中,假设bPtr指向数组开头)。就像数组元素可以用指针表达式引用一样,下列地址:

```
&b[ 3 ]
```

可以写成指针表达式:

```
bPtr + 3
```

数组本身可以当作指针并在指针算法中使用。例如，下列表达式：

```
*( b + 3)
```

同样引用数组元素 `b[3]`。一般来说，所有带下标的数组表达式都可以写成指针加偏移量，这时使用指针/偏移量符号，用数组名作为指针。注意，上述语句不修改数组名，`b` 还是指向数组中第一个元素。

指针和数组一样可以加下标。例如，下列表达式：

```
bPtr[ 1 ]
```

指数组元素 `b[1]`，这个表达式称为指针/下标符号（`pointer/subscript notation`）。

记住，数组名实际上是个常量指针，总是指向数组开头。因此下列表达式：

```
b += 3
```

是无效的，因为该表达式试图用指针算法修改数组名的值。

#### 常见编程错误 5.14

尽管数组名是指向数组开头的指针，而指针可以在算术表达式中修改，但数组名不可以在算术表达式中修改，因为数组名实际上是个常量指针。

图 5.20 的程序用我们介绍的四种方法引用数组元素（数组下标、用数组名作为指针的指针/偏移量符号、指针下标和指针的指针/偏移量符号，打印数组的 4 个元素）。

要演示数组和指针的互换性，还可以看看程序 5.21 中的两个字符串复制函数 `copy1` 和 `copy2`。这两个函数都是将字符串复制到字符数组中。比较 `copy1` 和 `copy2` 的函数原型可以发现，函数基本相同（由于数组和指针具有互换性）。这些函数完成相同的任务，但用不同方法实现。

```
1 // Fig. 5.20: fig05_20.cpp
2 // Using subscripting and pointer notations with arrays
3
4 #include <iostream.h>
5
6 int main()
7 {
8     int b[] = { 10, 20, 30, 40 };
9     int *bPtr = b;    // set bPtr to point to array b
10
11     cout << "Array b printed with:\n"
12         << "Array subscript notation\n";
13
14     for ( int i = 0; i < 4; i++ )
15         cout << "b[ " << i << " ] = " << b[ i ] << '\n';
16
17
18     cout << "\nPointer/offset notation where\n"
19         << "the pointer is the array name\n";
20
21     for ( int offset = 0; offset < 4; offset++ )
22         cout << "*" (b + " << offset << ") = "
```

```
23         << *( b + offset ) << '\n';
24
25
26     cout << "\nPointer subscript notation\n";
27
28     for ( i = 0; i < 4; i++ )
29         cout << "bPtr[ " << i << " ] = " << bPtr[ i ] << '\n';
30
31     cout << "\nPointer/offset notation\n";
32
33     for ( offset = 0; offset < 4; offset++ )
34         cout << "**(bPtr + " << offset << ") = "
35             << *( bPtr + offset ) << '\n';
36
37     return 0;
38 }
```

**输出结果:**

Array b Printed with:  
Array subscript notation  
b[ 0] = 10  
b[ 1] = 20  
b[ 2] = 30  
b[ 3] = 40

Pointer/offset notation where  
the pointer is the array name  
\*(b + 0) = 10  
\*(b + 1) = 20  
\*(b + 2) = 30  
\*(b + 3) = 40

Pointer subscript notation  
bPtr[ 0] = 10  
bPtr[ 1] = 20  
bPtr[ 2] = 30  
bPtr[ 3] = 40

Pointer/offset notation  
\*(bPtr + 0) = 10  
\*(bPtr + 1) = 20  
\*(bPtr + 2) = 30  
\*(bPtr + 3) = 40

图 5.20 用我们介绍的四种方法引用数组元素

```
1 // Fig. 5.21: fig05_21.cpp
2 // Copying a string using array notation
3 // and pointer notation.
4 #include <iostream.h>
5
6 void copy1( char *, const char * );
7 void copy2( char *, const char * );
8
9 int main()
```

```
10 {
11     char string1[ 10 ], *string2 = "Hello",
12         string3[ 10 ], string4[] = "Good Bye";
13
14     copy1( string1, string2 );
15     cout << "string1 = " << string1 << endl;
16
17     copy2( string3, string4 );
18     cout << "string3 = " << string3 << endl;
19
20     return 0;
21 }
22
23 // copy s2 to s1 using array notation
24 void copy1( char *s1, const char *s2 )
25 {
26     for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
27         ; // do nothing in body
28 }
29
30 // copy s2 to s1 using pointer notation
31 void copy2( char *s1, const char *s2 )
32 {
33     for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
34         ; // do nothing in body
35 }
```

**输出结果：**

```
string1 = Hello
string3 = Good Bye
```

图 5.21 使用数组和指针符号复制字符串

函数 `copy1` 用数组下标符号将 `s2` 中的字符串复制到字符数组 `s1` 中。函数声明一个作为数组下标的整型计数器变量 `i`。for 结构的首部进行整个复制操作，而 for 结构体本身是个空结构。首部中指定 `i` 初始化为 0，并在每次循环时加 1。for 的条件 “`(s1[i] = s2[i]) != '\0'`” 从 `s2` 向 `s1` 一次一个字符地进行复制操作。遇到 `s2` 中的 null 终止符时，将其赋给 `s1`，循环终止，因为 null 终止符等于 `'\0'`。记住，赋值语句的值是赋给左边参数的值。

函数 `copy2` 用指针和指针算法将 `s2` 中的字符串复制到 `s1` 字符数组。同样是在 for 结构的首部进行整个复制操作，首部没有任何变量初始化。和 `copy1` 中一样，条件 `(*s1 = *s2) != '\0'` 进行复制操作。复引用指针 `s2`，产生的字符赋给复引用的指针 `s1`。进行条件中的赋值之后，指针分别移到指向 `s1` 数组的下一个元素和字符串 `s2` 的下一个字符。遇到 `s2` 中的 null 终止符时，将其赋给 `s1`，循环终止。

注意 `copy1` 和 `copy2` 的第一个参数应当是足够大的数组，应能放下第二个参数中的字符串，否则可能会在写入数组边界以外的内存地址时发生错误。另外，注意每个函数中的第二个参数声明为 `const char *`（常量字符串）。在两个函数中，第二个参数都复制到第一个参数，一次一个地从第二个参数复制字符，但不对其做任何修改。因此，第二个参数声明为常量值的指针，实施最低权限原则。两个函数都不需要修改第二个参数，因此不向这两个函数提供修改第二个参数的功能。

## 5.9 指针数组

数组可以包含指针。这种数据结构的常见用法是构成字符串数组,通常称为字符串数组(string array)。字符串数组中的每项都是字符串,但在C++中,字符串实际上是第一个字符的指针。因此,字符串数组中的每项实际上是字符串中第一个字符的指针。下列字符串数组suit的声明可以表示一副牌:

```
char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades"};
```

声明的suit[ 4 ]部分表示4个元素的数组。声明的char \* 部分表示数组suit的每个元素是char类型的指针。数组中的4个值为"Hearts"、"Diamonds"、"Clubs"和"Spades"。每个值在内存中存放成比引号中的字符数多一个字符的null终止字符串。4个字符串长度分别为7、9、6、7。尽管这些字符串好像是放在suit数组中,其实数组中只存放指针(如图5.22)。每个指针指向对应字符串中的第一个字符。这样,尽管suit数组是定长的,但可以访问任意长度的字符串。这是C++强大的数据结构功能所带来的灵活性。

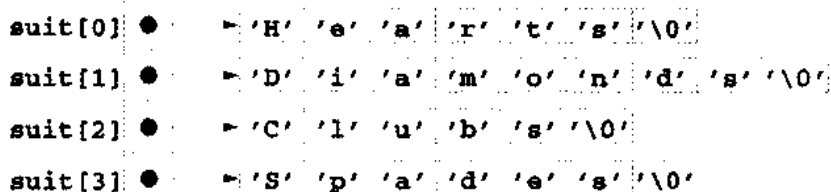


图 5.22 suit 数组的图形表示

suit字符串可以放在双下标数组中,每一行表示一个suit,每一列表示suit名的第一个字符。这种数据结构每一行应有固定列数,能够放下最长的字符串。因此,存放大量字符串而大部分字符串长度均比最长字符串短许多时,可能浪费很多内存空间。我们将在下一节用字符串数组帮助整理一副牌。

## 5.10 实例研究：洗牌与发牌

本节用随机数产生器开发一个洗牌与发牌程序。这个程序可以用于实现玩某种牌的游戏程序。为了解决一些微妙的性能问题,我们故意用次优洗牌与发牌算法。练习中要开发更有效的算法。

利用自上而下逐步完善的方法,我们开发一个程序,洗52张牌并发52张牌。自上而下逐步完善的方法在解决大而复杂的问题时特别有用。

我们用 $4 \times 13$ 的双下标数组deck表示要玩的牌(如图5.23)。行表示花色,0表示红心,1表示方块,2表示梅花,3表示黑桃。列表示牌的面值,0到9对应A到10,10到12对应J、Q、K。我们要装入字符串数组suit,用字符串表示4个花色,用字符串数组face的字符串表示13张牌的面值。

这堆牌可以进行如下的洗牌:首先将数组deck清空,然后随机选择row(0~3)和column(0~12)。将数字插入数组元素deck[ row ][ column ]表示这个牌是洗出的牌中要发的第一张牌)。继续这个过程,在deck数组中随机插入数字2、3、...、52,表示洗出的牌中要发的第二、三、...、五十二张牌。在deck数组填上牌号时,一张牌可能选择两次,即选择的时候deck[ row ][ column ]为非0值。



忽略这个选择, 随机重复选择其他 row 和 column, 直到找出未选择的牌。最后, 在 52 个 deck 元素中插入 1 到 52 的值。这时, 就完全洗好了牌。

		Ace	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Jack	Queen	King
		0	1	2	3	4	5	6	7	8	9	10	11	12
Hearts	0													
Diamonds	1													
Clubs	2													
Spades	3													

$\text{deck}[2][12]$  表示梅花 k  
 Clubs King

图 5.23 双下标数组 deck 表示要玩的牌

这个洗牌算法在随机重复选择已经洗过的牌时可能需要无限长的时间。这种现象称为无穷延迟 (indefinite postponement)。练习中将介绍更好的洗牌算法, 消除无穷延迟。

#### 性能提示 5.3

有时自然方式的算法可能包含无穷延迟等微妙的性能问题, 应寻找能避免无穷延迟的算法。

要发第一张牌, 我们要寻找匹配 1 的  $\text{deck}[\text{row}][\text{column}]$  元素, 这是用嵌套 for 结构进行的, row 取 0 到 3, column 取 0 到 12。这个数组元素对应哪种牌呢? suit 数组预先装入了四种花色, 因此要取花色, 只要打印字符串  $\text{suit}[\text{row}]$ ; 同样, 要取牌值, 只要打印字符串  $\text{face}[\text{column}]$ 。还要打印字符串 "of", 按正确的顺序打印, 即可得到每张牌如 "King of Clubs"、"Ace of Diamonds" 等等。

下面用自上而下逐步完善的方法进行。顶层为:

```
Shuffle and deal 52 cards
```

第一步完善结果为:

```
Initialize the suit array
Initialize the face array
Initialize the deck array
Shuffle the deck
Deal 52 cards
```

"Shuffle the deck" 可以展开成:

```
For each of the 52 cards
    Place card number in randomly selected unoccupied slot of deck
```

"Deal 52 cards" 可以展开成:

```
For each of the 52 cards
    Find card number in deck array and print face and suit of card
```

合在一起得到第二步完善结果为：

```
Initialize the suit array
Initialize the face array
Initialize the deck array

For each of the 52 cards
    Place card number in randomly selected unoccupied slot of deck

For each of the 52 cards
    Find card number in deck array and print face and suit of card
```

“Place card number in randomly selected unoccupied slot of deck” 可以展开成：

```
Choose slot of deck randomly

While chosen slot of deck has been previously chosen
    Choose slot of deck randomly

Place card number in chosen slot of deck
```

“Find card number in deck array and print face and suit of card” 可以展开成：

```
For each slot of the deck array
    If slot contains card number
        Print the face and suit of the card
```

合在一起得到第三步完善结果为：

```
Initialize the suit array
Initialize the face array
Initialize the deck array

For each of the 52 cards
    Choose slot of deck randomly

While slot of deck has been previously chosen
    Choose slot of deck randomly

    Place card number in chosen slot of deck

For each of the 52 cards
    For each slot of deck array
        If slot contains desired card number
            Print the face and suit of the card
```

这样就完成了完善过程。注意，如果将洗牌与发牌算法组合成每张牌在放到牌堆上进行发牌，则这个程序能更加有效。我们选择分别编程这些操作，因为通常是先洗后发，而不是边洗边发。

图5.24显示了洗牌与发牌程序，图5.25显示了示例执行结果。注意函数deal中使用的输出格式：

```
cout << setw( 5 ) << setiosflags( ios::right )
    << wFace[ column ] << " of "
    << setw( 8 ) << setiosflags( ios::left )
    << wSuit[ row ]
    << ( card % 2 == 0 ? '\n' : '\t' );
```

上述输出语句使牌的面值在5个字符的域中右对齐输出,而花色在8个字符的域中左对齐输出。输出打印成两列格式。如果输出的牌在第一列,则在后面输出一个制表符,移到第二列,否则输出换行符。

```

1 // Fig. 5.24: fig05_24.cpp
2 // Card shuffling and dealing program
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6 #include <time.h>
7
8 void shuffle( int[][ 13 ] );
9 void deal( const int[][ 13 ], const char *[], const char *[] );
10
11 int main()
12 {
13     const char *suit[ 4 ] =
14         { "Hearts", "Diamonds", "Clubs", "Spades" };
15     const char *face[ 13 ] =
16         { "Ace", "Deuce", "Three", "Four",
17           "Five", "Six", "Seven", "Eight",
18           "Nine", "Ten", "Jack", "Queen", "King" };
19     int deck[ 4 ][ 13 ] = { 0 };
20
21     srand( time( 0 ) );
22
23     shuffle( deck );
24     deal( deck, face, suit );
25
26     return 0;
27 }
28
29 void shuffle( int wDeck[][ 13 ] )
30 {
31     int row, column;
32
33     for ( int card = 1; card <= 52; card++ ) {
34         do {
35             row = rand() % 4;
36             column = rand() % 13;
37         } while( wDeck[ row ][ column ] != 0 );
38
39         wDeck[ row ][ column ] = card;
40     }
41 }
42
43 void deal( const int wDeck[][ 13 ], const char *wFace[],
44           const char *wSuit[] )
45 {
46     for ( int card = 1; card <= 52; card++ )
47         for ( int row = 0; row <= 3; row++ )
48             for ( int column = 0; column <= 12; column++ )
49
50
51

```

```

52         if ( wDeck[ row ][ column ] == card )
53             cout << setw( 5 ) << setiosflags( ios::right )
54                 << wFace[ column ] << " of "
55                 << setw( 8 ) << setiosflags( ios::left )
56                 << wSuit[ row ]
57                 << ( card % 2 == 0 ? '\n' : '\t' );
58 }

```

图 5.24 洗牌与发牌程序

输出结果:

Six of Clubs	Seven of Diamonds
Ace of Spades	Ace of Diamonds
Ace of Hearts	Queen of Diamonds
Queen of Clubs	Seven of Hearts
Ten of Hearts	Deuce of Clubs
Ten of Spades	Three of Spades
Ten of Diamonds	Four of Spades
Four of Diamonds	Ten of Clubs
Six of Diamonds	six of Spades
Eight of Hearts	Three of Diamonds
Nine of Hearts	Three of Hearts
Deuce of Spades	six of Hearts
Five of Clubs	Eight of Clubs
Deuce of Diamonds	Eight of Spades
Five of Spades	King of Clubs
King of Diamonds	Jack of Spades
Deuce of Hearts	Queen of Hearts
Ace of Clubs	King of Spades
Three of Clubs	King of Hearts
Nine of Clubs	Nine of Spades
Four of Hearts	Queen of Spades
Eight of Diamonds	Nine of Diamonds
Jack of Diamonds	Seven of Clubs
Five of Hearts	Five of Diamonds
Four of Clubs	Jack of Hearts
Jack of Clubs	Seven of Spades

图 5.25 洗牌与发牌程序的执行结果

发牌算法中有个缺点,一旦找到匹配之后,即使第一次就找到,两个内层的 for 结构仍然继续搜索 deck 中的其余元素。练习中要纠正这个缺点。

## 5.11 函数指针

函数指针包含函数在内存中的地址。第4章介绍了数组名实际上是数组中第一个元素的内存地址。同样,函数名实际上是执行函数任务的代码在内存中的开始地址。函数指针可以传入函数、从函数返回、存放在数组中和赋给其他的函数指针。

要演示如何使用函数指针,我们修改图 5.15 的冒泡排序程序,变成图 5.26 的程序。新程序包括 main 和函数 bubble、swap、ascending 和 descending。函数 bubbleSort 接收 ascending 或 descending 函数的函数指针参数以及一个整型数组和数组长度。程序提示用户选择按升序或降序排序。如果用

户输入1, 则向函数 bubble 传递 ascending 函数的指针, 使数组按升序排列。如果用户输入2, 则向函数 bubble 传递 descending 函数的指针, 使数组按降序排列。图 5.27 显示了示例的执行结果。

```
1 // Fig. 5.26: fig05_26.cpp
2 // Multipurpose sorting program using function pointers
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 void bubble( int [], const int, int (*)( int, int ) );
7 int ascending( int, int );
8 int descending( int, int );
9
10 int main()
11 {
12     const int arraySize = 10;
13     int order,
14         counter,
15         a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
16
17     cout << "Enter 1 to sort in ascending order,\n"
18          << "Enter 2 to sort in descending order: ";
19     cin >> order;
20     cout << "\nData items in original order\n";
21
22     for ( counter = 0; counter < arraySize; counter++ )
23         cout << setw( 4 ) << a[ counter ];
24
25     if ( order == 1 ) {
26         bubble( a, arraySize, ascending );
27         cout << "\nData items in ascending order\n";
28     }
29     else {
30         bubble( a, arraySize, descending );
31         cout << "\nData items in descending order\n";
32     }
33
34     for ( counter = 0; counter < arraySize; counter++ )
35         cout << setw( 4 ) << a[ counter ];
36
37     cout << endl;
38     return 0;
39 }
40
41 void bubble( int work[], const int size,
42             int (*compare)( int, int ) )
43 {
44     void swap( int *, int * );
45
46     for ( int pass = 1; pass < size; pass++ )
47
48         for ( int count = 0; count < size - 1; count++ )
49
50             if ( (*compare)( work[ count ], work[ count + 1 ] ) )
51                 swap( &work[ count ], &work[ count + 1 ] );
52 }
```

```

53
54 void swap( int *element1Ptr, int *element2Ptr )
55 {
56     int temp;
57
58     temp = *element1Ptr;
59     *element1Ptr = *element2Ptr;
60     *element2Ptr = temp;
61 }
62
63 int ascending( int a, int b )
64 {
65     return b < a;    // swap if b is less than a
66 }
67
68 int descending( int a, int b )
69 {
70     return b > a;    // swap if b is greater than a
71 }

```

图 5.26 使用函数指针的多用途排序程序

**输出结果：**

```

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
 2   6   4   8   10  12  89   68   45   37
Data items in ascending order
 2   4   6   8   10  12  37   45   68   89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
 2   6   4   8   10  12  89   68   45   37
Data items in ascending order
89   68   45   37   12  10   8    6    4    2

```

图 5.27 使用函数指针的多用途排序程序的执行结果

bubble 函数的首部中出现下列参数：

```
int ( *compare ) ( int, int )
```

告诉 bubble 该参数为一个函数指针，这个函数接收两个整型参数和返回一个整型值。`*compare` 周围要用括号，因为 `*` 的优先级低于函数参数周围所用括号的优先级。如果不用括号，则声明变成：

```
int *compare ( int, int )
```

声明函数接收两个整型参数并返回一个整型值的指针。

bubble 函数原型中的对应参数如下所示：

```
int ( * )( int, int )
```

注意这里只包括类型，程序员可以加上名称，但参数名只用于程序中的说明，编译器将其忽略。

if 语句中调用传入 bubble 的函数，如下所示：

```
if ( ( *compare )( work[ count ], work[ count + 1 ] ) )
```

就像复引用变量指针可以访问变量值一样，复引用函数指针可以执行这个函数。

也可以不复引用指针而调用函数，如下所示：

```
if ( compare( work[ count ], work[ count + 1 ] ) )
```

直接用指针作为函数名。我们更愿意使用第一种通过指针调用函数的方法，因为它显式说明 compare 是函数指针，通过复引用指针而调用这个函数。第二种通过指针调用函数的方法使 compare 好像是个实际函数。程序用户可能搞糊涂，想看看 compare 函数的定义却怎么也找不到。

函数指针的一个用法是建立菜单驱动系统，提示用户从菜单选择一个选项（例如从 1 到 5）。每个选项由不同函数提供服务，每个函数的指针存放在函数指针数组中。用户选项作为数组下标，数组中的指针用于调用这个函数。

图 5.28 的程序提供了声明和使用函数指针数组的一般例子。这些函数（function1、function2 和 function3）都定义成取整数参数并且不返回值。这些函数的指针存放在数组 f 中，声明如下：

```
void ( *f[ 3 ] )( int ) = { function1, function2, function3 }
```

声明从最左边的括号读起，表示 f 是 3 个函数指针的数组，各取整数参数并返回 void。数组用三个函数名（是指针）初始化。用户输入 0 到 2 的值时，用这些值作为函数指针数组的下标。函数调用如下所示：

```
( *f[ choice ] )( choice );
```

调用时，f[ choice ] 选择数组中 choice 位置的指针。复引用指针以调用函数，并将 choice 作为参数传入函数中。每个函数打印自己的参数值和函数名，表示正确调用了这个函数。练习中要开发一个菜单驱动系统。

```
1 // Fig. 5.28: fig05_28.cpp
2 // Demonstrating an array of pointers to functions
3 #include <iostream.h>
4 void function1( int );
5 void function2( int );
6 void function3( int );
7
8 int main()
9 {
10     void ( *f[ 3 ] )( int ) = { function1, function2, function3 };
11     int choice;
12
13     cout << "Enter a number between 0 and 2, 3 to end: ";
14     cin >> choice;
15
16     while ( choice >= 0 && choice < 3 ) {
17         ( *f[ choice ] )( choice );
18         cout << "Enter a number between 0 and 2, 3 to end: ";
19         cin >> choice;
20     }
21 }
```

```
22     cout << "Program execution completed." << endl;
23     return 0;
24 }
25
26 void function1( int a )
27 {
28     cout << "You entered " << a
29         << " so function1 was called\n\n";
30 }
31
32 void function2( int b )
33 {
34     cout << "You entered " << b
35         << " so function2 was called\n\n";
36 }
37
38 void function3( int c )
39 {
40     cout << "You entered " << c
41         << " so function3 was called\n\n";
42 }
```

**输出结果：**

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called
```

```
Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called
```

```
Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called
```

```
Enter a number between 0 and 2, 3 to end: 3
Program execution completed
```

图 5.28 声明和使用函数的指针数组

## 5.12 字符与字符串处理简介

本节要介绍一些字符串处理的标准库函数。这里介绍的技术适用于开发文本编辑器、字处理器、桌面排版软件、计算机化打字系统和其他文本处理软件。我们这里使用基于指针的字符串，本书稍后还将介绍把字符串作为成熟的对象。

### 5.12.1 字符与字符串基础

字符是C++编程语言的基本组件。每个程序都是由一系列字符用有意义的方式组合而成的，计算机将其解释为一系列指令，用来完成一组任务。程序可能包含字符常量（character constant）。字符常量是表示为单引号字符的整数值。字符常量的值是机器字符集中该字符的整数值。例如，'z'表示z的整数值（在ASCII字符集中为122），'\n'表示换行符的整数值（在ASCII字符集中为10）。



字符串就是把一系列字符当作一个单元处理。字符串可能包含字母、数字和+、-、\*、/、\$等各种特殊字符（special character）。C++中的字符串直接量（string literal）或字符串常量（string constant）放在双引号中如下所示：

"John Q. Doe"	（姓名）
"9999 Main Street"	（街道）
"Waltham, Massachusetts"	（州）
"(201) 555-1212"	（电话号码）

C++中的字符串是以null终止符（'\0'）结尾的字符数组。通过字符串中第一个字符的指针来访问字符串。字符串的值是字符串中第一个字符的地址（常量），这样，C++中可以说字符串是个常量指针，是指向字符串中第一个字符的指针。从这个意义上说，字符串像数组一样，因为数组名也是第一个元素的（常量）指针。

可以在声明中将字符串指定为字符数组或char\*类型的变量。下列声明：

```
char color[] = "blue";  
char *colorPtr = "blue";
```

分别将变量初始化为"blue"。第一个声明生成5个元素的数组color，包含字符'b'、'l'、'u'、'e'和'\0'。第二个声明生成指针变量colorPtr，指向内存中的字符串"blue"。

#### 可移植性提示 5.5

用字符串直接量初始化char\*类型的变量时，有些编译器将字符串放在内存中无法修改字符串的位置。如果要修改字符串直接量，则应将其存放在字符数组中，以便在所有系统中修改。

声明char color[] = {"blue"};也可以改写成：

```
char color[] = {'b', 'l', 'u', 'e', '\0'};
```

声明包含字符串的字符数组时，数组应足够大，能存放字符串及其null终止符。上述声明自动根据初始化值列表中提供的初始化值的个数确定数组长度。

#### 常见编程错误 5.15

字符数组中没有分配能存放字符串及其null终止符的足够空间。

#### 常见编程错误 5.16

生成或使用不包含null终止符的字符串。

#### 编程技巧 5.5

在字符数组中存放字符串时，一定要保证能存放要存的最长字符串。C++允许存放任意长度的字符串。如果字符串长度超过字符数组长度，则越界字符会改写数组后面的内存地址中存放的数据。

字符串可以用cin通过流读取赋给数组。例如，下列语句将字符串赋给字符数组word[20]：

```
cin >> word;
```

用户输入的字符串存放在word中。上述语句读取字符，直到遇到空格、制表符、换行符或文件结束符。注意，字符串不能超过19个字符，因为还要留下null终止符的空间。第2章介绍的setw流操纵算子可以用于保证读取到word的字符串不超过字符数组长度。例如，下列语句：

```
cin >> setw( 20 ) >> word;
```

指定 cin 最多读取 19 个字符到数组 word 中，并将数组第 20 个位置用于保存字符串的 null 终止符。setw 流操作算子只能用于输入的下一个值。

有时，可以将整行文本输入数组中。为此，C++ 提供了 cin.getline 函数。cin.getline 函数取三个参数：存放该行文本的字符数组、长度和分隔符。例如，下列程序段：

```
char sentence[ 80 ];  
cin.getline( sentence, 80, '\n' );
```

声明 80 个字符的数组 sentence，然后从键盘读取一行文本到该数组中。函数遇到分隔符 '\n'、输入文件结束符或读取的字符数比第二个参数的长度少 1 时停止读取字符（最后一个字符位置用于保存字符串的 null 终止符）。如果遇到分隔符，则读取该分隔符并不再读入下一字符。cin.getline 函数的第三个参数默认值为 '\n'，因此上述函数调用也可以改写成：

```
cin.getline( sentence, 80 );
```

第 11 章“C++ 输入/输出流”中将会详细介绍 cin.getline 和其他函数。

#### 常见编程错误 5.17

将单个字符作为字符串处理可能导致致命的运行时错误。字符串是指针，可能对应一个大整数。而单个字符是个小整数（0-255 的 ASCII 值）。在许多系统中，这会导致错误，因为低内存地址是用于特殊用途的，如操作系统中断处理器，因此会发生“访问无效”的错误。

#### 常见编程错误 5.18

需要字符串参数时将字符传入函数可能导致致命的运行时错误。

#### 常见编程错误 5.19

需要字符参数时将字符串传入函数是个语法错误。

### 5.12.2 字符串处理库的字符串操作函数

字符串处理库提供许多操作字符串数据、比较字符串、搜索字符串中的字符与其他字符串、将字符串标记化（将字符串分成各个逻辑组件）和确定字符串长度的字符串操作函数。本节介绍字符串处理库（标准库）中常用的字符串操作函数。图 5.29 总结了这些函数。

注意图 5.29 中的几个函数包含 size\_t 数据类型的参数。这是在头文件 <stddef.h>（标准库中的头文件，标准库中还有许多其他标准库头文件，包括 <string.h>）中定义为 unsigned int 或 unsigned long 之类的无符号整数类型。

#### 常见编程错误 5.20

使用字符串处理库中的函数而不包括 <string.h> 头文件。

函数 strcpy 将第二个参数（字符串）复制到第一个参数（字符数组）中，这个字符数组的长度应当足以放下字符串及其 null 终止符。函数 strncpy 与 strcpy 相似，只是 strncpy 指定从字符串复制到字符数组的字符数。注意函数 strncpy 不一定复制第二个参数的 null 终止符，null 终止符要在复制的字符数比字符串长度至少多 1 时才复制。例如，如果第二个参数为“test”，则只在 strncpy 的第三个参数至少为 5（“test”的长度加 null 终止符）时才复制 null 终止符。如果第三个参数大于 5，则数组后面添加 null 终止符，直到写入第三个参数指定的总字符数。

函数原型	函数说明
<code>char* strcpy( char* s1, const char* s2 )</code>	将字符串 s2 复制到字符数组 s1 中, 返回 s1 的值
<code>char* strncpy( char* s1, const char* s2, size_t n )</code>	将字符串 s2 中最多 n 个字符复制到字符数组 s1 中, 返回 s1 的值
<code>char* strcat( char* s1, const char* s2 )</code>	将字符串 s2 添加到字符串 s1 后面。s2 的第一个字符重定义 s1 的 null 终止符。返回 s1 的值
<code>char* strncat( char* s1, const char* s2, size_t n )</code>	将字符串 s2 中最多 n 个字符添加到字符串 s1 后面。s2 的第一个字符重定义 s1 的 null 终止符。返回 s1 的值
<code>int strcmp( const char* s1, const char* s2 )</code>	比较字符串 s1 与字符串 s2。函数在 s1 等于、小于或大于 s2 时分别返回 0、小于 0 或大于 0 的值
<code>int strncmp( const char* s1, const char* s2, size_t n )</code>	比较字符串 s1 中的 n 个字符与字符串 s2。函数在 s1 等于、小于或大于 s2 时分别返回 0、小于 0 或大于 0 的值
<code>char* strtok( char* s1, const char* s2 )</code>	用一系列 strtok 调用将 s1 字符串标记化(将字符串分成各个逻辑组件, 如同一行文本中的每个单词), 用字符串 s2 所包含的字符分隔。首次调用时包含 s1 为第一个参数, 后面调用时继续标记化同一字符串, 包含 NULL 为第一个参数。每次调用时返回当前标记的指针。如果函数调用时不再有更多标记, 则返回 NULL
<code>size_t strlen( const char* s )</code>	确定字符串长度, 返回 null 终止符之前的字符数

图 5.29 字符串处理库的字符串操作函数

## 常见编程错误 5.21

第三个参数小于或等于第二个参数的字符串长度时不在 strncpy 的第一个参数中添加 null 终止符可能造成严重的运行时错误。

图 5.30 的程序用 strcpy 将数组 x 中的整个字符串复制到数组 y 中, 并用 strncpy 将数组 x 的前 14 个字符复制到数组 z 中。将 null 字符 ('\0') 添加到数组 z, 因为程序中调用 strncpy 时没有写入 null 终止符(第三个参数小于或等于第二个参数的字符串长度)。

```

1 // Fig. 5.30: fig05_30.cpp
2 // Using strcpy and strncpy
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char x[] = "Happy Birthday to You";
9     char y[ 25 ], z[ 15 ];
10
11     cout << "The string in array x is: " << x
12         << "\nThe string in array y is: " << strcpy( y, x )
13         << '\n';
14     strncpy( z, x, 14 ); // does not copy null character
15     z[ 14 ] = '\0';
16     cout << "The string in array z is: " << z << endl;
17
18     return 0;
19 }

```

**输出结果：**

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

图 5.30 使用 strcpy 和 strncpy 函数

函数 `strcat` 将第二个参数（字符串）添加到第一个参数（字符数组）中。第二个参数的第一个字符代替终止第一个参数中字符串的 null 终止符（'\0'）。程序员要保证存放第一个字符串的数组应足以存放第一个字符串、第二个字符串和 null 终止符（从第二个字符串复制）的合并长度。函数 `strncat` 从第二个字符串添加指定字符数到第一个字符串中，并在结果中添加 null 终止符。图 5.31 的程序演示了函数 `strcat` 和 `strncat`。

图 5.32 用 `strcmp` 和 `strncmp` 比较三个字符串。函数 `strcmp` 一次一个字符地比较第一个字符串参数与第二个字符串参数。如果字符串相等，则函数返回 0；如果第一个字符串小于第二个字符串，则函数返回负值；如果第一个字符串大于第二个字符串，则函数返回正值。函数 `strncmp` 等价于函数 `strcmp`，只是 `strncmp` 只比较到指定字符数。函数 `strncmp` 不比较字符串中 null 终止符后面的字符。程序打印每次函数调用返回的整数值。

**常见编程错误 5.22**

假设 `strcmp` 和 `strncmp` 在参数相等时返回 1 是个逻辑错误。`strcmp` 和 `strncmp` 在参数相等时返回 0（C++ 的 false 值）。因此测试两个字符串的相等性时，`strcmp` 和 `strncmp` 的结果应与 0 比较，确定字符串是否相等。

```
1 // Fig. 5.31: fig05_31.cpp
2 // Using strcat and strncat
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char s1[ 20 ] = "Happy ";
9     char s2[] = "New Year ";
10    char s3[ 40 ] = "";
11
12    cout << "s1 = " << s1 << "\ns2 = " << s2;
13    cout << "\nstrcat(s1, s2) = " << strcat( s1, s2 );
14    cout << "\nstrncat(s3, s1, 6) = " << strncat( s3, s1, 6 );
15    cout << "\nstrcat(s3, s1) = " << strcat( s3, s1 ) << endl;
16
17    return 0;
18 }
```

**输出结果：**

```
s1 = Happy
s2 = New Year
strcat(s1, s2) = Happy New Year
strncat(s3, s1, 6) = Happy
strcat(s3, s1) = Happy Happy New Year
```

图 5.31 使用 strcat 和 strncat 函数

```
1 // Fig. 5.32: fig05_32.cpp
2 // Using strcmp and strncmp
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <string.h>
6
7 int main()
8 {
9     char *s1 = "Happy New Year";
10    char *s2 = "Happy New Year";
11    char *s3 = "Happy Holidays";
12
13    cout << "s1 = " << s1 << "\ns2 = " << s2
14          << "\ns3 = " << s3 << "\n\nstrcmp(s1, s2) = "
15          << setw( 2 ) << strcmp( s1, s2 )
16          << "\nstrcmp(s1, s3) = " << setw( 2 )
17          << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
18          << setw( 2 ) << strcmp( s3, s1 );
19
20    cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
21          << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = "
22          << setw( 2 ) << strncmp( s1, s3, 7 )
23          << "\nstrncmp(s3, s1, 7) = "
24          << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;
25    return 0;
26 }
```

**输出结果:**

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays
```

```
strcmp(s1, s2) = 0
strcmp (s1, s3) = 1
strcmp (s3, s1) = -1
```

```
strncmp(s1, s3, 6) = 0
strncmp(s1, s3, 7) = 1
strncmp(s3, s1, 7) = -1
```

图 5.32 使用 strcmp 和 strncmp 函数

要了解一个字符串大于或小于另一字符串的含义,可以考虑一系列姓氏的字母顺序表。读者一定会把“Jones”放在“Smith”之前,因为“Jones”的第一个字母在“Smith”的第一个字母之前。但字母表中不仅有26个字母,而是个字母顺序表,每个字母在表中有特定位置。“Z”不仅表示字母,而且是字母表中第二十六个字母。

计算机怎么知道一个字母在另一字母之前呢?所有字符在计算机中均表示为数字代码,计算机比较两个字符串时,实际上是比较字符串中字符的数字代码。

**可移植性提示 5.6**

不同计算机上可能用不同的内部数字代码表示字符。

**可移植性提示 5.7**

不要显式测试 ASCII 码如 `if (ch == 65);`，而要用对应的字符常量，例如 `if (ch == 'A');`。

为了实现标准化字符表示，大多数计算机厂家将机器设计成使用两种常用编码系统：ASCII 和 EBCDIC。ASCII 指“美国标准信息交换码”（American Standard Code for Information Interchange），EBCDIC 指“扩展二进制编码的十进制交换码”（Extended Binary Coded Decimal Interchange Code）。还有其他编码系统，但这是两种最常用的编码系统。

ASCII 和 EBCDIC 称为字符编码（character code）或字符集（character set）。字符串和字符操作实际上是在操作相应的数字代码，而不是操作字符本身。因此 C++ 中字符和小整数具有互换性。由于数字代码之间有大干、等于、小于的关系，因此可以将不同字符或字符串通过字符编码相互比较。附录 B 列出了 ASCII 字符编码。

函数 `strtok` 将字符串分解为一系列标记（token）。标记就是一系列用分隔符（delimiting character，通常是空格或标点符号）分开的字符。例如，在一行文本中，每个单词可以作为标记，空格是分隔符。

需要多次调用 `strtok` 才能将字符串分解为标记（假设字符串中包含多个标记）。第一次调用 `strtok` 包含两个参数，即要标记化的字符串和包含用来分隔标记的字符的字符串（即分隔符）。在图 5.33 的例子中，下列语句：

```
tokenPtr = strtok( string, " " );
```

将 `tokenPtr` 赋给 `string` 中第一个标记的指针。`strtok` 的第二个参数 `" "` 表示 `string` 中的标记用空格分开。函数 `strtok` 搜索 `string` 中不是分隔符（空格）的第一个字符，这是第一个标记的开头。然后函数寻找字符串中的下一个分隔符，将其换成 `null ( '\0' )` 字符，这是当前标记的终点。函数 `strtok` 保存 `string` 中标记后面的下一个字符的指针，并返回当前标记的指针。

后面再调用 `strtok` 时，第一个参数为 `NULL`，继续将 `string` 标记化。`NULL` 参数表示调用 `strtok` 继续从 `string` 中上次调用 `strtok` 时保存的位置开始标记化。如果调用 `strtok` 时已经没有标记，则 `strtok` 返回 `NULL`。图 5.33 的程序用 `strtok` 将字符串 `"This is a sentence with 7 tokens"` 标记化。分别打印每个标记。注意 `strtok` 修改输入字符串，因此，如果调用 `strtok` 之后还要在程序中使用这个字符串，则应复制这个字符串。

**常见编程错误 5.23**

没有认识到 `strtok` 修改正在标记化的字符串，调用 `strtok` 之后还在程序中使用这个字符串（以为还是原字符串）。

函数 `strlen` 取一个字符串作为参数，并返回字符串中的字符个数，长度中不包括 `null` 终止符。图 5.34 的程序演示了函数 `strlen`。

```
1 // Fig. 5.33: fig05_33.cpp
2 // Using strtok
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char string[] = "This is a sentence with 7 tokens";
9     char *tokenPtr;
10
11     cout << "The string to be tokenized is:\n" << string
12         << "\n\nThe tokens are:\n";
```

```
13
14     tokenPtr = strtok( string, " " );
15
16     while ( tokenPtr != NULL ) {
17         cout << tokenPtr << '\n';
18         tokenPtr = strtok( NULL, " " );
19     }
20
21     return 0;
22 }
```

**输出结果:**

The string to be tokenized is:  
This is a sentence with 7 tokens

The tokens are:

This  
is  
a  
sentence  
with  
7  
tokens

图 5.33 使用 strtok 函数

```
1 // Fig. 5.34: fig05_34.cpp
2 // Using strlen
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "abcdefghijklmnopqrstuvwxyz";
9     char *string2 = "four";
10    char *string3 = "Boston";
11
12    cout << "The length of \"" << string1
13         << "\" is " << strlen( string1 )
14         << "\nThe length of \"" << string2
15         << "\" is " << strlen( string2 )
16         << "\nThe length of \"" << string3
17         << "\" is " << strlen( string3 ) << endl;
18
19    return 0;
20 }
```

**输出结果:**

The length of "abcdefghijklmnopqrstuvwxyz" is 26  
The length of "four" is 4  
The length of "Boston" is 6

图 5.34 使用 strlen 函数

## 5.13 有关对象的思考：对象间的交互

这是第6章开始介绍C++面向对象编程之前的最后一次面向对象设计任务。完成这个任务并学习第6章之后，就可以开始编写电梯模拟程序。要完成第2章定义的电梯模拟程序，就需要第6章、第7章和第8章介绍的C++方法。然后用第9章和第10章介绍的继承和多态方法修改电梯模拟程序。

本节介绍对象间的交互帮助读者把对象联系起来。也许读者在第2章、第3章和第4章开发实验室任务时增加了对象、属性和行为。

我们介绍过，大多数C++对象不是自动工作的，而是响应消息刺激，实际上是对象对成员函数的函数调用。

下面要考虑电梯模拟程序中几个类的交互。问题“人按楼层中的按钮”，这里主语是人，宾语是按钮，这就是类之间交互的例子。person类的对象向button类的对象发一个消息，我们把这个消息称为pushButton。上一章把该消息变成了button类的成员函数。

这时，在电梯模拟程序每个类的其他事实中，余下的大概都是类之间的交互，其中有些明显表示类对象之间的交互。但请考虑下列表述：

`"person waits for elevator door to open"`(人等待电梯开门)

上一章列出了电梯门的两个行为，称为openDoor和closeDoor，现在要确定哪个类对象发出这些消息。可以确定是电梯本身发出这些消息给电梯门。这些类对象之间的交互隐含在问题陈述中。

现在要继续细化电梯模拟程序中每个类的“其他事实”部分，这时应当主要包含类之间的交互。注意下列几点：

1. 谁发送类对象。
2. 发送什么消息。
3. 谁接收类对象。

在每个类中，增加“发往其他类对象的消息”（这种消息也称为“协作”）并列类之间的其他交互，例如，在person类中，包含项目：

`person sends pushButton message to the button on that floor`

在button类的“协作”部分，放上消息：

`button sends comeGetMe message to elevator`

建立这些项目时，可以增加对象的属性和行为。

完成这个实验室练习后，就有了实现电梯模拟程序的完整类清单。对每个类，有了完整的属性和行为清单以及对象之间要发送的消息清单。

下一章要开始介绍C++面向对象编程，介绍如何生成新类。学习第6章之后，就可以用C++编写电梯模拟程序的重要部分；学习第7章和第8章之后，就可以实现可运行的电梯模拟程序；学习第9章和第10章之后，就可以使用继承，利用类之间的共性减少解决问题所需的编程工作量。

下面小结第2章到第5章介绍的面向对象设计过程。

1. 在文本文件中输入问题陈述。



2. 放弃不重要的文本。
3. 取出所有事实，在事实文件中每一行列出一个事实。
4. 找出事实中的名词，其通常是需要的类，对每个类进行高度概括。
5. 对每个事实，放在相应类中进行二级概述。如果一个事实涉及几个类，则放在涉及的每个类中。
6. 组分类中的事实，列出属性、行为和协作。
7. 在属性中，列出与每个类相关的数据。
8. 在行为中，列出每个类对象收到消息时进行的操作。每个行为是类的成员函数。
9. 在协作中，列出这个类对象向其他类对象发出的消息和接收该消息的类对象。
10. 这时设计中可能还缺点什么，第6章介绍如何用C++实现电梯模拟程序时，这些问题会显露出来。

属性与行为通常称为类的“责任”。这里介绍的设计方法有时称为类、责任与协作(简单CRC)。

## 小结

- 指针变量的值为内存地址。
- 下列声明：

```
int *ptr
```

声明变量 ptr 为 int 类型对象的指针，或者说成“ptr 是 int 的指针”。声明为指针的每个变量前面都要加上星号 (\*)。

- 指针可以初始化为 0、NULL 或一个地址。数值为 0 或 NULL 的指针不指任何内容。
- 数值 0 是可以直接赋给指针变量的惟一整数值。
- & (地址) 运算符返回操作数的地址。
- 地址运算符的操作数必须是变量名，地址运算符不能用于常量、不产生引用的表达式和用存储类 register 声明的变量。
- \* 运算符通常称为间接运算符或复引用运算符。返回操作数 (即指针) 所指对象的同义词、别名或浑名。这种使用 \* 的方法称为复引用指针。
- 调用函数并要修改参数时，传递该参数地址。被调用函数使用间接运算符 (\*) 修改调用函数中的参数值。
- 接收地址参数的函数要定义接收地址的指针参数。
- 不需要在函数原型中包括指针名，只要包括指针类型即可。参数名仅用于程序中的说明，编译器将其忽略。
- const 限定符使程序员通知编译器，特定变量的值不能修改。
- 如果试图修改 const 类型的值，则编译器会捕获这个错误并发出一个警告或错误消息 (取决于特定的编译器)。
- 将指针传递给函数有四种方法：非常量数据的非常量指针、非常量数据的常量指针、非常量数据的常量指针和常量数据的常量指针。
- 数组自动用指针按引用传递，因为数组名的值为数组地址。
- 要用指针按引用调用传递单个数组元素，就必须传递特定数组元素的地址。

- C++ 提供一元运算符 `sizeof`，确定程序执行期间的数组长度或其他数据类型长度（字节数）。
- 采用数组名时，`sizeof` 运算符返回数组中的总字节数为 `size_t` 类型的值，通常是 `unsigned int` 类型。
- `sizeof` 运算符可以用于任何变量名、类型名或常量值。
- 指针可以使用自增（++）或自减（--）运算符，整数可以加进指针中（+ 或 +=），也可以从指针中减去整数（- 或 -=），指针可以减去另一指针。
- 将指针增加或减去一个整数时，指针并不是直接增加或减去这个整数，而是加上指针所指对象长度的这个倍数。
- 对相邻内存地址（如数组）进行指针算法时才有意义。数组中的所有元素在内存中是相邻存放的。
- 对字符数组进行指针算法时，结果与普通算法相同，因为每个字符占一个字节内存。
- 如果两个指针的类型相同，则可以将一个指针赋给另一个指针。否则要用强制类型转换运算符将赋值语句右边的指针值转换为赋值语句左边的指针值。这个规则的例外是 `void` 的指针（即 `void*`），该指针是个一般性指针，可以表示任何指针类型。所有指针类型都可以赋给 `void` 指针而不需要进行强制类型转换。但是，`void` 指针不能直接赋给另一类型的指针，而要先将 `void` 指针转换为正确的指针类型。
- `void*` 指针不能复引用。
- 指针可以用相等和关系运算符比较，但这种比较只在对相同数组成员进行时才有意义。
- 指针也可以像数组一样带有下标。
- 数组名等同于数组第一个元素的指针。
- 指针 / 偏移量符号中的偏移量等同于数组下标。
- 所有带下标的数组表达式都可以写成指针和偏移量，或是用数组名作为指针，或是用一个独立的指针指向数组。
- 数组名实际上是个常量指针，总是指向数组开头的内存地址。
- 数组可以包含指针。
- 函数指针包含函数在内存中的地址。
- 函数指针可以传入函数、从函数返回、存放在数组中和赋给其他的函数指针。
- 函数指针的一个用法是建立菜单驱动系统，提示用户从菜单选择一个选项。
- 函数 `strcpy` 将第二个参数（字符串）复制到第一个参数（字符数组）中，这个字符数组的长度应当足以放下字符串及其 `null` 终止符。
- 函数 `strncpy` 与 `strcpy` 相似，只是 `strncpy` 指定从字符串复制到字符数组的字符数。注意函数 `strncpy` 不一定复制第二个参数的 `null` 终止符，`null` 终止符要在复制的字符数比字符串长度至少多 1 时才复制。
- 函数 `strcat` 将第二个参数（字符串）添加到第一个参数（字符数组）中。第二个参数的第一个字符代替终止第一个参数中字符串的 `null` 终止符（'\0'）。程序员要保证存放第一个字符串的数组应足以存放第一个字符串和第二个字符串。
- 函数 `strncat` 从第二个字符串添加指定字符数到第一个字符串中，并在结果中添加 `null` 终止符。
- 函数 `strcmp` 一次一个字符地比较第一个字符串参数与第二个字符串参数。如果字符串相等，则函数返回 0；如果第一个字符串小于第二个字符串，则函数返回负值；如果第一个字符串大于第二个字符串，则函数返回正值。

# 原书缺页

string constant 字符串常量	strtok
string literal 字符串直接量	subtracting an integer from a pointer 将指针减去一个整数
string processing 字符串处理	subtracting two pointers 两个指针相减
string.h	token 标记
strlen	tokenizing strings 标记化字符串
strncat	void * (pointer to void) void 指针
strcmp	word processing 字处理
strcpy	

## 自测练习

### 5.1 填空

- 指针变量包含另一变量的 \_\_\_\_\_ 值。
- 可以初始化指针的值有 \_\_\_\_\_、\_\_\_\_\_ 或 \_\_\_\_\_。
- 可以赋给指针的惟一整数是 \_\_\_\_\_。

### 5.2 判断下列各题是否正确。如果不正确，请说明原因。

- 地址运算符 & 只能用于常量、表达式和 register 存储类声明的变量。
- 声明为 void 的指针可以复引用。
- 不同类型的指针不必进行强制类型转换操作即可相互赋值。

### 5.3 回答下列问题。假设单精度浮点数存放在4字节中，数组在内存中的开始地址为1002500。每道习题尽量使用前面的结果。

- 声明 float 类型数组 numbers，含有 10 个元素，初始化为 0.0、1.1、2.2...9.9。假设符号化常量 SIZE 定义为 10。
- 声明指针 nPtr，指向 float 类型对象。
- 用数组下标符号打印数组 numbers 的元素。使用 for 结构，假设声明了整型控制变量 i。打印每个数，小数点后面的精度为 1。
- 用两条不同语句将数组 numbers 的开始地址赋给指针变量 nPtr。
- 用指针 nPtr 和指针 / 偏移量符号打印数组 numbers 的元素。
- 用数组名作为指针和指针 / 偏移量符号打印数组 numbers 的元素。
- 用指针 nPtr 的下标打印数组 numbers 的元素。
- 用数组下标符号、数组名作为指针和指针 / 偏移量符号、nPtr 与指针下标符号和 nPtr 与指针 / 偏移量符号打印数组 numbers 的元素 4。
- 假设 nPtr 指向数组 numbers 开头，nPtr + 8 指哪个地址？这个地址存放什么值？
- 假设 nPtr 指向 numbers[5]，执行 nPtr -= 4 之后 nPtr 引用哪个地址？这个地址存放什么值？

### 5.4 对下列各题，各编写一条语句。假设已声明浮点数变量 number1 和 number2，number1 初始化为 7.3。并假设变量 ptr 为 char\* 类型，数组 s1[100] 和 s2[100] 为 char 类型。

- 声明变量 fPtr 为 float 类型对象的指针。
- 将变量 number1 的地址赋给指针变量 fPtr。
- 打印 fPtr 所指的对象的值。
- 指定 fPtr 所指对象值为变量 number2。

- e) 打印 number2 的值。
- f) 打印 number1 的地址。
- g) 打印 iPtr 中存放的地址, 打印的值是否与 number1 的地址相同?
- h) 将数组 s2 中存放的字符串复制到数组 s1 中。
- i) 比较 s1 中的字符串与 s2 中的字符串并打印结果。
- j) 将 s2 中字符串中的 10 个字符添加到 s1 中的字符串中。
- k) 确定 s1 中的字符串的长度。
- l) 将 s2 中第一个标记的地址赋给 ptr。S2 中的标记用逗号 (,) 分开。

#### 5.5 根据题目要求编写语句。

- a) 编写函数 exchange 的函数首部, 取两个浮点数 x 和 y 的指针为参数, 不返回数值。
- b) 编写 a) 中函数的函数原型。
- c) 编写函数 evaluate 的函数首部, 返回整数, 取整数 x 和函数 poly 的指针参数。函数 poly 取一个整数参数并返回一个整数。
- d) 编写 c) 中函数的函数原型。
- e) 显示用元音字符串 "AEIOU" 初始化字符数组 vowel 的两种不同方法。

#### 5.6 找出下列程序段中的错误。假设:

```
int *zPtr;    //zPtr will reference array z
int aPtr = 0;
void *sPtr = 0;
int number, i;
int z[ 5 ] = { 1, 2, 3, 4, 5 };
sPtr = z;
```

- a) ++zPtr;
- b) // use pointer to get first value of array  
number = zPtr;
- c) // assign array element 2 (the value 3) to number  
number = \*zPtr[ 2 ];
- d) // print entire array z  
for ( i = 0; i <= 5; i++ )  
 cout << zPtr[ i ] << endl;
- e) // assign the value pointed to by sPtr to number  
number = sPtr;
- f) ++z;
- g) char s[ 10 ];  
 cout << strncpy( s, "hello", 5 ) << endl;
- h) char s[ 12 ];  
 strcpy( s, "Welcome Home" );
- i) if ( strcmp( string1, string2 ))  
 cout << "The strings are equal" << endl;

#### 5.7 执行下列语句时打印什么 (如果有)? 如果语句中有错, 说明错误及纠正方法。假设声明下列变量:

```
char s1[ 50 ] = "jack", s2[ 50 ] = "jill", s3[ 50 ], *sPtr;
```

- a) cout << strcpy( s3, s2 ) << endl;
- b) cout << strcat( strcat( strcpy( s3, s1 ), " and " ), s2 )  
 << endl;

```
c) cout << strlen( s1 ) + strlen( s2 ) << endl;
d) cout << strlen( s3 ) << endl;
```

### 自测练习答案

- 5.1 a) 地址。b) 0、NULL 或地址。c) 0。
- 5.2 a) 不正确。地址运算符只能用于变量，不能用于常量、表达式或用存储类 register 声明的变量。  
 b) 不正确。void 的指针无法复引用，因为无法知道要用多少内存字节复引用。  
 c) 不正确。void 类型指针可以赋给其他类型的指针。void 类型指针要通过显式地强制类型转换才可以赋给其他类型的指针。
- 5.3 a) `float numbers[ SIZE ] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };`  
 b) `float *nPtr;`  
 c) `cout << setiosflags( ios::fixed | ios::showpoint )`  
     `<< setprecision( 1 );`  
     `for ( i = 0; i < SIZE; i++ )`  
         `cout << numbers[ i ] << ' ';`  
 d) `nPtr = numbers;`  
     `nPtr = &numbers[ 0 ];`  
 e) `cout << setiosflags( ios::fixed | ios::showpoint )`  
     `<< setprecision( 1 );`  
     `for ( i = 0; i < SIZE; i++ )`  
         `cout << *( nPtr + i ) << ' ';`  
 f) `cout << setiosflags( ios::fixed | ios::showpoint )`  
     `<< setprecision( 1 );`  
     `for ( i = 0; i < SIZE; i++ )`  
         `cout << *( numbers + i ) << ' ';`  
 g) `cout << setiosflags( ios::fixed | ios::showpoint )`  
     `<< setprecision( 1 );`  
     `for ( i = 0; i < SIZE; i++ )`  
         `cout << *( numbers + i ) << ' ';`  
 h) `numbers[ 4 ]`  
     `*( numbers + 4 )`  
     `nPtr[ 4 ]`  
     `*( nPtr + 4 )`  
 i) 地址为  $1002500 + 8 * 4 = 1002532$  值为 8.8。  
 j) `numbers[ 5 ]` 的地址为  $1002500 + 5 * 4 = 1002520$ 。  
     `nPtr -= 4` 的地址为  $1002500 - 4 * 4 = 1002504$   
     该地址的值为 1.1。
- 5.4 a) `float *fPtr;`  
 b) `fPtr = &number1;`  
 c) `cout << "The value of * fPtr is " << *fPtr << endl;`  
 d) `number2 = *fPtr;`  
 e) `cout << "The value of number2 is " << number2 << endl;`  
 f) `cout << "The address of number1 is " << number1 << endl;`  
 g) `cout << "The address stored in fPtr is " << fPtr << endl;`  
     相同。  
 h) `strcpy(s1,s2);`  
 i) `cout << "strcmp(s1,s2) = " << strcmp(s1,s2) << endl;`

- ```
j) strcat( s1, s2, 10 );
k) cout << "strlen(s1) = " << strlen( s1 ) << endl;
l) ptr = strtok( s2, "," );
```
- 5.5 a) void exchange( float \*, float \*y )  
 b) void exchange( float \*x, float \* );  
 c) int evaluate( int x, int (\*poly) ( int ) )  
 d) int evaluate( int,int (\*)( int ) );  
 e) char vowel [] = "AEIOU";  
     char vowel [] = { 'A', 'E', 'I', 'O', 'U', '\0' };
- 5.6 a) 不正确: zPtr 没有初始化。  
     纠正: 用 zPtr = z; 初始化 zPtr。  
 b) 不正确: 指针没有复引用。  
     纠正: 将该语句变成 number = \*zPtr;  
 c) 不正确: zPtr[ 2 ]不是指针, 不能复引用。  
     纠正: 将 zPtr[ 2 ]变为 \*zPtr[ 2 ]  
 d) 不正确: 指针下标引用数组界限之外的数组元素。  
     纠正: 将for结构中的关系运算符变为“<”以避免指针下标引用数组界限之外的数组元素。  
 e) 不正确: void 指针无法复引用。  
     纠正: 要复引用指针, 首先要将其转换为整型指针。将上述语句变为:  
     number = \*(int \*)sPtr;  
 f) 不正确: 指针算法修改数组名。  
     纠正: 用指针变量而不用数组名完成指针算法,或在数组名后面加上下标引用特定元素。  
 g) 不正确: 函数 strcpy 没有将 null 终止符写入数组 s, 因为第三个参数等于字符串 "hello" 的长度。  
     纠正: 将 strcpy 的第三个参数变为 6 或对 s[ 5 ]赋值 '\0', 确保在字符串后加上终止 null 符。  
 h) 不正确: 字符数组 s 太小, 不能存放 null 终止符。  
     纠正: 声明更多元素的数组。  
 i) 不正确: 函数 strcmp 在字符串相等时返回 0, 因此 if 结构中的条件为假, 不执行输出语句。  
     纠正: 在 if 结构条件中将 strcmp 的结果与 0 比较。
- 5.7 a) jill  
     b) jack and jill  
     c) 8  
     d) 13

## 练习

- 5.8 判断对错, 并说明原因。  
 a) 比较指向两个不同数组的指针是没有意义的。  
 b) 由于数组名是指向数组第一个元素的指针, 因此数组名可以和指针一样进行操作。
- 5.9 回答下列问题。假设无符号整数存放在 2 字节中, 数组的开始内存地址为 1002500。  
 a) 声明 5 个元素的 unsigned int 类型数组 values, 并将其元素初始化为 2 到 10 的偶数, 假设已经将符号化常量 SIZE 定义为 5。

- b) 声明指针 `vPtr`, 指向 `unsigned int` 类型的对象。
  - c) 用数组下标符号打印数组 `values` 的元素。使用 `for` 结构, 并假设已经声明整型控制变量 `i`。
  - d) 用两个不同语句将数组 `values` 的开始地址指定为指针变量 `vPtr`。
  - e) 用指针 / 偏移量符号打印数组 `values` 的元素。
  - f) 用数组名作为指针和指针 / 偏移量符号打印数组 `values` 的元素。
  - g) 用数组指针的下标打印数组 `values` 的元素。
  - h) 用数组下标符号、数组名作为指针和指针 / 偏移量符号、指针下标符号和指针 / 偏移量符号引用 `values` 的元素 5。
  - i) `vPtr + 3` 引用什么地址? 该地址存放什么值?
  - j) 假设 `vPtr` 指向 `values[4]`, `vPtr--` 指向什么地址, 该地址存放什么值?
- 5.10 对下列各题, 各编写一条语句。假设已声明长整型变量 `value1` 和 `value2`, `value1` 初始化为 200000。
- a) 声明变量 `lPtr` 为 `long` 类型对象的指针。
  - b) 将变量 `value1` 的地址赋给指针变量 `lPtr`。
  - c) 打印 `lPtr` 所指的对象值。
  - d) 指定 `lPtr` 所指对象值为变量 `value2`。
  - e) 打印 `value2` 值。
  - f) 打印 `value1` 地址。
  - g) 打印 `lPtr` 中存放的地址, 打印的值是否与 `value1` 的地址相同?
- 5.11 根据题目要求编写语句。
- a) 编写函数 `zero` 的函数首部, 取长整数数组参数 `bigIntegers`, 不返回数值。
  - b) 写出 a) 中函数的函数原型。
  - c) 编写函数 `add1AndSum` 的函数首部, 取整数数组参数 `oneTooSmall` 并返回一个整数值。
  - d) 写出 c) 中函数的函数原型。
- 说明:** 练习 5.12 到 5.15 比较难。完成这些练习后, 就可以很容易地实现常见的扑克牌游戏了。
- 5.12 修改图 5.24 的程序, 使洗牌函数向牌手发五张牌, 然后编写完成下列任务的函数:
- a) 确定手中是否有一对牌。
  - b) 确定手中是否有对牌。
  - c) 确定手中是否有三色同号牌 (如三张 J)。
  - d) 确定手中是否有四色同号牌 (如四张 A)。
  - e) 确定手中是否有同花 (即五张牌花色相同)。
  - f) 确定手中是否有一条龙 (即五张牌牌号连续)。
- 5.13 用练习 5.12 建立的函数编写一个程序, 发两手五张牌, 确定两手牌哪个更好。
- 5.14 修改 5.13 练习中的程序, 模拟发牌器。发牌器的五张牌是盖起来的, 游戏者看不到。然后程序求值这手牌, 根据牌的质量, 抓一张、两张或三张牌, 换掉原来手中不要的牌。然后程序重新求值这手牌。注意: 这是个难题。
- 5.15 修改练习 5.14 的程序, 使其能自动处理发牌器中的牌, 但游戏者可以确定自己手中要换的牌。然后程序求值两手牌, 确定谁赢。用这个新程序与计算机玩 20 把, 看看是你赢还是计算机赢。再让你的朋友与计算机玩 20 把, 看看谁赢得多。根据这些游戏的结果, 完



善扑克游戏程序（又是个难题）。再与计算机玩20把，修改后的程序是否能够玩更好的游戏？

- 5.16 在图 5.24 的洗牌与发牌程序中，我们故意用无效的洗牌算法，引入无穷延迟的概念。在这个练习中，要生成高性能的洗牌算法，避免无穷延迟。

按照下面的做法修改图 5.24。初始化 deck 数组（如图 5.35）。修改 shuffle 函数，在数组中一行一行、一列一列地循环，到达每个元素一次。每个元素与随机选择的数组元素进行交换。打印结果数组，确定是否洗好了牌（如图 5.36）。程序可能要多次调用 shuffle 函数，才能洗好牌。

注意，尽管这个练习改进了洗牌算法，但洗牌算法仍然要从 deck 数组搜索第 1 张牌、第 2 张牌、第 3 张牌等等。更糟的是，即使在发牌算法找到并发出牌之后，该算法仍然搜索牌堆中的其他元素。修改图 5.24 的程序，使发牌之后不再继续匹配这张牌，程序立即转入发下一张牌。

|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 1 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 2 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 3 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 |

图 5.35 未洗过牌的 deck 数组

| 0 | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 19 | 40 | 27 | 25 | 36 | 46 | 10 | 34 | 35 | 41 | 18 | 2  | 44 |
| 1 | 13 | 28 | 14 | 16 | 21 | 30 | 8  | 11 | 31 | 17 | 24 | 7  | 1  |
| 2 | 12 | 33 | 15 | 42 | 43 | 23 | 45 | 3  | 29 | 32 | 4  | 47 | 26 |
| 3 | 50 | 38 | 52 | 39 | 48 | 51 | 9  | 5  | 37 | 49 | 22 | 6  | 20 |

图 5.36 洗过牌的 deck 数组示例

- 5.17（模拟龟兔赛跑）本练习中要模拟龟兔赛跑的寓言故事。用随机数产生器建立模拟龟兔赛跑的程序。对手从 70 个方格的第 1 格开始起跑，每格表示跑道上的一个可能位置，终点线在第 70 格处。第一个到达终点的选手奖励一个新鲜萝卜和莴苣。兔子要在山坡上睡一觉，因此可能失去冠军。

有一个每秒钟滴答一次的钟，程序应按下列规则调整动物的位置：

| 动物            | 运动类型            | 时间百分比 | 实际运动    |
|---------------|-----------------|-------|---------|
| 乌龟 (Tortoise) | Fast plod (快走)  | 50%   | 向右 3 格  |
|               | Slip (摔跤)       | 20%   | 向左 6 格  |
|               | Slow plod (慢走)  | 30%   | 向右 1 格  |
| 兔子 (Hare)     | Sleep (睡觉)      | 20%   | 不动      |
|               | Big hop (大跳)    | 20%   | 向右 9 格  |
|               | Big slip (大跌)   | 10%   | 向左 12 格 |
|               | Small hop (小跳)  | 30%   | 向右 1 格  |
|               | Small slip (小跌) | 20%   | 向左 2 格  |

用变量跟踪动物的位置（即位置号 1 到 70）。每个动物从位置 1 开始，如果动物跌到第 1 格以外，则移回第 1 格。

产生随机整数  $i$  ( $1 \leq i \leq 10$ )，以得到上表中的百分比。对于乌龟， $1 \leq i \leq 5$  时快走， $6 \leq i \leq 7$  时跌跤， $8 \leq i \leq 10$  时慢走，兔子也用相似的方法。

起跑时，打印：

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```

时钟每次滴答一下（即每个重复循环），打印第 70 格位置的一条线，显示乌龟的位置 T 和兔子的位置 H。如果两者占用一格，则乌龟会咬兔子，程序从该位置开始打印 OUCH!!!。除 T、H 和 OUCH!!! 以外的其他打印位置都是空的。

打印每一行之后，测试某个动物是否超过了第 70 格，如果是，则打印获胜者，停止模拟。如果乌龟赢，则打印 TORTOISE WINS!!! YAY!!!。如果兔子赢，则打印 Hare wins. Yush。如果两个动物同时赢，则可以同情弱者，让乌龟赢，或者打印 It's a tie。如果两者都没有赢，则再次循环，模拟下一个时钟滴答。准备运行程序时，让一组拉拉队看比赛，你会发现观众有多么投入。

### 特殊小节：建立自己的计算机

下面几个问题要暂时离开高级语言编程，打开计算机，看看其内部结构。我们介绍机器语言编程和编写几个机器语言程序。要让这些知识更有价值，我们建立一个计算机（通过软件模拟技术），在其中执行我们的机器语言程序。

- 5.18（机器语言程序）下面要建立一个 Simpletron 计算机。顾名思义，这是个简单机器，但以后会发现它也是个强大的机器。Simpletron 只能运行用 Simpletron Machine Language(SML 机器语言)写成的程序。

Simpletron 包含一个累加器（特殊寄存器），存放 Simpletron 用于计算和各种处理的信息。Simpletron 处理的所有信息都用“字”处理。字是个带符号的四位十进制数，如 +3364，-1293，+0007，-0001 等等。Simpletron 带有 100 个字的内存，这些字用其内存单元号 00、01、...99 引用。

运行 SML 程序之前，要先把程序装入内存。每个 SML 程序中的第一条指令（或语句）一般放在内存单元 00 处，模拟器会从该地址开始执行。

用 SML 编写的每条指令都占用 Simpletron 内存中的一个字。（因此，指令是带符号的四位十进制数）。我们应假设 SML 指令的符号总是正号，但数据字的符号可正可负。Simpletron 内存中的每个内存单元可以包含一条指令、程序使用的数据值或未用（未定义）内存区。每个 SML 指令的前两位是操作码，指定要进行的操作。图 5.37 显示了 SML 操作码。

SML 指令的最后两位是操作数，是要操作的字的特定内存单元。

下面考虑几个简单 SML 程序。第一个 SML 程序（例 1）从键盘读取两个数，并计算和打印这两个数的和。指令 +1007 从键盘读取第一个数并将其放在内存单元 07（初始化为 0），然后指令 +1008 读取下一个数并将其放在内存单元 08。装入命令 +2007 将第一个数放（复制）到累加器中，加法指令 +3008 将第二个数与累加器中的数相加。所有 SML

算术运算指令都把结果留在累加器中。保存指令+2109将结果复制回内存单元09, 然后写指令+1109取得并打印这个结果(带符号的四位十进制数), 停止指令+4300终止程序执行。

| 操作码                        | 意义                            |
|----------------------------|-------------------------------|
| 输入/输出操作:                   |                               |
| const int READ = 10        | 从键盘读一个字到特定内存单元                |
| const int WRITE = 11;      | 从特定内存单元写一个字到屏幕                |
| 装入/保存操作:                   |                               |
| const int LOAD = 20;       | 从特定内存单元将字装入累加器                |
| const int STORE = 21;      | 将累加器中的字存放到特定内存单元              |
| 算术运算:                      |                               |
| const int ADD = 30;        | 将特定内存单元中的字加上累加器中的字(结果保留在累加器中) |
| const int SUBTRACT = 31;   | 将累加器中的字减去特定内存单元中的字(结果保留在累加器中) |
| const int DIVIDE = 32;     | 将累加器中的字除以特定内存单元中的字(结果保留在累加器中) |
| const int MULTIPLY = 33;   | 将特定内存单元中的字乘以累加器中的字(结果保留在累加器中) |
| 控制转移操作:                    |                               |
| const int BRANCH = 40;     | 转移到特定内存单元                     |
| const int BRANCHNEG = 41;  | 在累加器为负值时转移到特定内存单元             |
| const int BRANCHZERO = 42; | 在累加器为0时转移到特定内存单元              |
| const int HALT = 43;       | 停止, 程序已完成任务                   |

图 5.37 SML 机器语言操作码

| 例 1 地址 | 数值    | 指令             |
|--------|-------|----------------|
| 00     | +1007 | ( Read A )     |
| 01     | +1008 | ( Read B )     |
| 02     | +2007 | ( Load A )     |
| 03     | +3008 | ( Add B )      |
| 04     | +2109 | ( Store C )    |
| 05     | +1109 | ( Write C )    |
| 06     | +4300 | ( Halt )       |
| 07     | +0000 | ( Variable A ) |
| 08     | +0000 | ( Variable B ) |
| 09     | +0000 | ( Result C )   |

例2中的SML程序从键盘读取两个数, 并确定和打印其中较大的数。注意这里用指令+4107作为条件控制转移, 与C++中的if语句相似。

| 例 2 地址 | 数值    | 指令                        |
|--------|-------|---------------------------|
| 00     | +1009 | ( Read A )                |
| 01     | +1010 | ( Read B )                |
| 02     | +2009 | ( Load A )                |
| 03     | +3110 | ( Subtract B )            |
| 04     | +4107 | ( Branch negative to 07 ) |
| 05     | +1109 | ( Write A )               |
| 06     | +4300 | ( Halt )                  |
| 07     | +1110 | ( Write B )               |
| 08     | +4300 | ( Halt )                  |
| 09     | +0000 | ( Variable A )            |
| 10     | +0000 | ( Variable B )            |

现在编写完成下列任务的 SML 程序：

- a) 用标记控制循环读取 10 个正数值，计算和打印它们的和。
- b) 用计数器控制循环读取 7 个数，有正有负，计算和打印它们平均值。
- c) 读取一系列数，并确定和打印其最大数。第一个读取的数表示要处理多少个数。

5.19 (计算机模拟程序) 这里要建立一台计算机，当然不是硬件连接，而是用软件模拟，建立 Simpletron 的软件模型。这个 Simpletron 模拟程序可以将读者的计算机变成 Simpletron 并可以实际运行。测试和调试练习 5.18 所编写的 SML 程序。

这个 Simpletron 模拟程序运行时，开始打印：

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?). ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program. ***
```

用 100 个元素的单下标数组 memory 模拟 Simpletron 内存。然后假设运行这个 Simpletron 模拟程序，检查一下练习 5.18 例 2 的程序：

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Program loading completed ***
*** Program execution begins ***
```

这时 SML 程序已放进数组 memory 中，Simpletron 开始执行编写的 SML 程序。程序从内存单元 00 的指令开始执行，和 C++ 中一样，按顺序继续执行，但可以通过控制转移转入程序的其他部分。

利用变量 accumulator 表示累加寄存器。用变量 counter 跟踪内存中包含所执行指令的内存单元。用变量 operationCode 表示当前正在进行的操作，即指令字的左边两位。用变量 operand 表示操作当前指令的内存单元即指令字的右边两位。不要直接从内存中直接执行指令，而是将下一个要执行的指令从内存中转到变量 instructionRegister 中。然后选取左边两位，放进 operationCode 中，选取右边两位，放进 operand 中。Simpletron 开始执行时，所有特殊寄存器都初始化为 0。

下面看看第一个 SML 指令（内存地址 00 的指令 +1009）的执行过程，这是条指令执行循环（instruction execution cycle）。

counter 指示下一个要执行指令的内存单元。我们用下列 C++ 语句从 memory 中取得该地址的内容：

```
instructionRegister = memory[ counter ];
```

下列语句从指令寄存器读取操作码和操作数：

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

现在 Simpletron 必须确定操作码是读（而不是写、装入等等）。switch 结构区分 SML 的十二种操作。

在 switch 结构中，各种 SML 操作指令的模拟如下（其他留给读者练习）：

```
读 (read):           cin >> memory[ operand ];
装入 (load):         accumulator = memory[ operand ];
加 (add):             accumulator += memory[ operand ];
转移 (branch):       稍后介绍转移
停止 (halt):         这条指令打印下列消息
                      ***Simpletron excution terminated ***
```

然后打印每个寄存器的名称与内容即内存的完整内容。这种打印输出称为计算机转储 (computer dump)。为了帮助编制转储功能，图 5.38 显示了一个示例转储格式。注意执行 Simpletron 程序之后，计算机转储显示指令实际值和终止执行时的数据值。

下面继续执行我们程序的第一条指令，内存单元 00 中的 +1009。前面曾介绍过，switch 语句用下列 C++ 语句模拟这个过程：

```
cin >> memory[ operand ];
```

执行 cin 之前，屏幕上显示一个问号(?)，提示用户输入。Simpletron 等待用户输入一个值并按 Return 键。然后将这个值读取到内存单元 09。

这时，第一条指令模拟已经完成。余下的就是准备让 Simpletron 执行下一条指令。由于刚刚完成的指令不是控制转移，因此只要增加指令计数器寄存器，如下所示：

```
++counter;
```

这就完成了第一条指令的模拟执行。整个过程（即指令执行循环）重新开始，读取下一个要执行的指令。

现在考虑如何模拟分支结构（控制转移），只要调整指令计数器的值即可。因此，无条件转移指令（40）可以用 switch 模拟如下：

```
counter = operand;
```

条件（累加器为 0 则转移）指令模拟如下：

```
if ( accumulator == 0 )
    counter = operand;
```

这时就可以实现 Simpletron 模拟程序和运行练习 5.18 中编写的每个程序了。可以在 SML 中增加其他特性，并在 Simpletron 模拟程序中提供这些特性。

Simpletron 模拟程序应检查各种错误。例如，程序装入期间，用户输入 Simpletron 程序的 memory 中的每个数应在 -9999 到 +9999 之间。Simpletron 模拟程序应当用 while 循环测试输入的每个数值是否在这个范围中，如果不是，则提示用户重新输入，直到输入正确的值。

```
REGISTERS:
accumulator      +0000
counter          00
instructionRegister +0000
operationCode     00
operand          00
MEMORY:
      0      1      2      3      4      5      6      7      8      9
0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

图 5.38 示例转储

在执行过程中，Simpletron 模拟程序应检查各种严重错误，如除数为 0、执行无效操作码、累加器溢出（即算术运算的结果不在 -9999 到 +9999 之间）等等。这种严重错误称为致命错误。发现致命错误时，Simpletron 模拟程序应打印下列错误消息：

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

并按前面介绍的格式打印完整的计算机转储，这样可以帮助用户找到程序中的错误。

## 更多的指针练习

5.20 修改图 5.24 的洗牌与发牌程序，使洗牌与发牌操作由同一函数(shuffleAndDeal)完成。这个函数应包含嵌套循环结构，类似于图 5.24 中的函数 shuffle。

5.21 下列程序有什么作用？

```
1 // ex05_21.cpp
2 #include <iostream.h>
3
4 void mystery1( char *, const char * );
5
6 int main()
7 {
8     char string1[ 80 ], string2[ 80 ];
9
10    cout << "Enter two strings: ";
11    cin >> string1 >> string2;
12    mystery1( string1, string2 );
```

```

13  cout << string1 << endl;
14  return 0;
15 }
16
17 void mystery1( char *s1, const char *s2 )
18 {
19     while ( *s1 != '\0' )
20         ++s1;
21
22     for ( ; *s1 = *s2; s1++, s2++ )
23         ;    // empty statement
24 }

```

### 5.22 下列程序有什么作用?

```

1 // ex05_22.cpp
2 #include <iostream.h>
3
4 void mystery2( const char * );
5
6 int main()
7 {
8     char string[ 80 ];
9
10    cout << "Enter two strings: ";
11    cin >> string;
12    cout << mystery2( string ) << endl;
13    return 0;
14 }
15
16 int mystery2( const char *s )
17 {
18     for ( int x = 0; *s != '\0'; s++ )
19         ++x;
20
21     return x;
22 }

```

### 5.23 找出下列程序中的错误。如果能纠正, 请说明纠正方法。

- a) `int *number;`  
`cout << number << endl;`
- b) `float *realPtr;`  
`long *integerPtr;`  
`integerPtr = realPtr;`
- c) `int * x, y;`  
`x = y;`
- d) `char s[] = "this is a character array";`  
`for ( ; *s != '\0'; s++)`  
`cout << *s << ' ';`
- e) `short * numPtr, result;`  
`void *genericPtr = numPtr;`  
`result = *genericPtr + 7;`
- f) `float x = 19.34;`  
`float xPtr = &x;`  
`cout << xPtr << endl;`
- g) `char *s;`  
`cout << s << endl;`

5.24 (快速排序) 在第4章的例子和练习中, 我们介绍了冒泡排序、桶排序和选择排序。现在要介绍称为快速排序的递归排序方法。单下标数组值的基本算法如下:

a) 分区步骤: 取未排序数组的第一个元素, 确定其在排序数组中的最终位置, 即该元素左边的所有值小于该元素, 该元素右边的所有值大于该元素。这样就确定了一个元素位置, 有了两个未排序子数组。

b) 递归步骤: 对每个未排序子数组完成第一步。

每次对未排序子数组完成第一步时, 又确定了一个元素位置, 有了另外两个未排序子数组。当子数组只有一个元素时, 就已经排序完毕, 该元素已经在最终位置了。

基本算法似乎很简单, 但如何确定每个子数组的第一个元素在排序数组中的最终位置呢? 例如, 考虑下列数值(黑体元素是分区元素, 要放在排序数组中的最终位置)

**37** 2 6 4 89 8 10 12 68 45

a) 从数组最右边的元素开始, 比较 **37** 与每个元素, 直到找出小于 **37** 的元素, 然后将这个元素与 **37** 交换。第一个小于 **37** 的元素是 12, 因此将 12 与 **37** 交换。新数组如下:

12 2 6 4 89 8 10 **37** 68 45

元素 12 用斜体表示刚刚与 37 交换。

b) 从数组左边 12 以后的元素开始, 比较 **37** 与每个元素, 直到找出大于 **37** 的元素, 然后将这个元素与 **37** 交换。第一个大于 **37** 的元素是 89, 因此将 89 与 **37** 交换。新数组如下:

12 2 6 4 **37** 8 10 89 68 45

c) 从数组右边 89 以前的元素开始, 比较 **37** 与每个元素, 直到找出小于 **37** 的元素, 然后将这个元素与 **37** 交换。第一个小于 **37** 的元素是 10, 因此将 10 与 **37** 交换。新数组如下:

12 2 6 4 10 8 **37** 89 68 45

d) 从数组左边 10 以后的元素开始, 比较 **37** 与每个元素, 直到找出大于 **37** 的元素, 然后将这个元素与 **37** 交换。由于没有比 **37** 更大的元素, 因此 **37** 已经放在排序数组中的最终位置。

对上述数组采用分区后, 就出现两个未排序小数组。小于 **37** 的未排序小数组包含 12、2、6、4、10 和 8。大于 **37** 的未排序小数组包含 89、68 和 45。对这两个未排序小数组进行像原数组一样的处理。

根据上述介绍, 编写一个递归函数 quickSort, 排序单下标整型数组。函数接收一个整型数组、开始下标和结束下标参数。quickSort 调用函数 partition 函数进行分区。

5.25 (走迷宫) 下列 # 和圆点(.)组成的网格是表示迷宫的双下标数组。

```
# # # # # # # # # # #
# . # . # . # . # . #
# # # . # . # . # . #
# . # . # . # . # . #
# # # # . # . # . # .
# . # . # . # . # . #
# # . # . # . # . # .
# . # . # . # . # . #
# # # # . # . # . # .
# # # # # # . # . # .
# . # . # . # . # . #
# # # # # # # # # # #
```



上述双下标数组中，#表示迷宫的墙，圆点表示可以走的路线，只能在数组中包含圆点的地方移动。

走迷宫的一个简单算法总能走到出口（如果有）。如果没有出口，则会回到起始点。将右手放在右边的墙上并开始前进，手不离墙，最终总能走到出口。当然，可能还有更短的路径，但上述路径总能走到出口。

编写走迷宫的递归函数 `mazeTraverse`。函数接收  $12 \times 12$  字符的数组，表示这个迷宫，并接收开始位置参数。`mazeTraverse` 在寻找迷宫的出口时，应将字符 `x` 放在沿途的每一格。每次移动之后，函数应显示迷宫，让用户能看到迷宫的走法。

- 5.26（随机产生迷宫）编写一个 `mazeGenerator` 函数，接受  $12 \times 12$  字符的数组并随机产生迷宫。函数还应提供迷宫的开始和结束位置。试用随机产生迷宫测试练习 5.25 所写的函数 `mazeTraverse`。
- 5.27（任何大小的迷宫）将练习 5.25 和 5.26 的函数 `mazeTraverse` 与 `mazeGenerator` 一般化，可以处理任何大小的迷宫。
- 5.28（函数指针数组）将图 4.23 的程序改写成使用菜单驱动界面。程序提供 5 个选项如下所示（应在屏幕上显示）：

```
Enter a choice:
0 Print the array of grades
1 Find the minimum grade
2 Find the maximum grade
3 Print the average on all tests for each student
4 End program
```

使用函数指针数组的一个限制是所有指针应为相同类型。指针应指向接收相同类型参数和返回相同类型数值的函数。因此，图 4.23 的函数应修改成接收相同类型参数和返回相同类型数值。将函数 `minimum` 和 `maximum` 修改成打印最小值与最大值，不返回任何内容。对选项 3，将图 4.23 的函数 `average` 修改成输出每个学生（而不是特定学生）的平均成绩。函数 `average` 与函数 `printArray`、`minimum` 和 `maximum` 接收相同类型参数且不返回任何内容。将 4 个函数的指针存放在数组 `processGrades` 中，并用用户选择的选项作为调用每个函数的数组下标。

- 5.29（修改 Simpletron 模拟程序）练习 5.19 中编写了计算机的软件模拟，执行用 Simpletron Machine Language (SML) 编写的程序。本练习中要对 Simpletron 模拟程序进行几个修改和补充。练习 15.26 和 15.27 中将建立一个编译器，将高级编程语言（BASIC 的变形）编写的程序转换为 SML。要执行编译器产生的程序，需要进行下列修改和补充。
- a) 将 Simpletron 模拟程序的内存扩展为包含 1000 个内存单元，使 Simpletron 模拟程序能处理更大的程序。
  - b) 让模拟程序进行求模计算，这要求增加 SML 指令。
  - c) 让模拟程序进行指数计算，这要求增加 SML 指令。
  - d) 将 Simpletron 模拟程序修改成用十六进制值而不是用整数值表示 SML 指令。
  - e) 将 Simpletron 模拟程序修改成允许输出换行符。这要求增加 SML 指令。
  - f) 将 Simpletron 模拟程序修改成不仅能处理整数值，而且能处理浮点数。
  - g) 将 Simpletron 模拟程序修改成处理字符串输入。提示：每个 Simpletron 字可以分为两组，各放两位整数。每个两位整数表示一个字符的 ASCII 十进制的对应值。增加在特定的

Simpletron 内存单元开始输入和存放字符串的机器语言指令。该内存单元的字的前半部分是字符串中的字符数(即字符串的长度),后半部分包含一个用两个十进制位所表示的 ASCII 字符。机器语言指令将每个字符变为对应 ASCII 值并赋给半个字。

- h) 将 Simpletron 模拟程序修改成处理字符串输出(按 g)中的格式存放)。提示:增加在特定 Simpletron 内存单元开始打印字符串的机器语言指令。该内存单元的字的前半部分是字符串中的字符数(即字符串的长度),后半部分包含一个用两个十进制位表示的 ASCII 字符。机器语言指令将每个两位数变为对应字符,检查字符串长度,并通过将两位数转换成相应字符来打印字符串。

### 5.30 下列程序有什么作用?

```
1 // ex05_30.cpp
2 #include <iostream.h>
3
4 int mystery3( const char *, const char * );
5
6 int main()
7 {
8     char string1[ 80 ], string2[ 80 ];
9
10    cout << "Enter two strings: ";
11    cin >> string1 >> string2;
12    cout << "The result is "
13         << mystery3( string1, string2 ) << endl;
14
15    return 0;
16 }
17
18 int mystery3 ( const char *s1, const char *s2 )
19 {
20     for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ )
21
22         if ( *s1 != *s2 )
23             return 0;
24
25     return 1;
26 }
```

## 字符串操作练习

- 5.31 编写一个程序,用函数 `strcmp` 比较用户输入的两个字符串。程序指出第一个字符串是小于、等于或大于第二个字符串。
- 5.32 编写一个程序,用函数 `strncmp` 比较用户输入的两个字符串,程序要输入比较的字符数。程序指出第一个字符串是小于、等于或大于第二个字符串。
- 5.33 编写一个程序,用随机数产生器建立语句。程序用 4 个 `char` 类型的指针数组 `article`、`noun`、`verb` 和 `preposition`。程序按下列顺序从 4 个数组分别随机取一个元素生成语句: `article`、`noun`、`verb`、`preposition`、`article` 和 `noun`。选择每个单词时,在能放下整个句子的数组中连接上述单词。单词之间用空格分开。输出最后的语句时,应以大写字母开头,以圆点结尾。程序产生 20 个句子。

数组填充如下: article 数组包含冠词 "the"、"a"、"one"、"some" 和 "any", noun 数组包含名词 "boy"、"girl"、"dog"、"town" 和 "car", verb 数组包含动词 "drove"、"jumped"、"ran"、"walked" 和 "skipped", preposition 数组包含介词 "to"、"from"、"over"、"under" 和 "on"。

编写上述程序之后, 将程序修改成产生由几个句子组成的短故事(这样就可以编写一篇自动文章)。

5.34 (五行打油诗) 五行打油诗由五句话组成, 第一行、第二行与第五行压韵, 第三行与第四行压韵。利用练习 5.33 介绍的方法, 编写一个随机产生五行打油诗的 C++ 程序。要产生好的五行打油诗并不容易, 但这个工作非常有趣。

5.35 编写一个将英语短语编成 pig Latin 的程序, pig Latin 就是故意打乱单词的字母顺序, 下面是一个简单的 pig Latin 算法。

要将英语短语编成 pig Latin, 用函数 strtok 将短语标记化为各个单词。要把单词变成 pig Latin, 将第一个字母放到末尾, 并加上 "ay" 字样, 如 "jump" 变成 "umpjay", "the" 变成 "hetay", "computer" 变成 "omputercay"。单词之间的空格保持不变。假设单词之间用空格分开, 没有标点符号, 每个单词均由两个以上字母组成。函数 printLatinWord 显示每个单词。提示: 每次调用 strtok 并找到一个标记时, 将标记指针传递给函数 printLatinWord, 并打印 pig Latin 单词。

5.36 编写一个程序, 以 (555) 555-5555 形式输入电话号码字符串。程序用函数 strtok 取得区号标记, 电话号码的前三位作为一个标记, 后四位作为另一个标记。电话号码的七位数连接成一个字符串。程序将区号字符串变为 int 型, 将电话号码字符串变为 long 型, 并打印区号和电话号码字符串。

5.37 编写一个程序, 输入一行文本, 用 strtok 函数标记化该行文本, 并以相反顺序输出标记。

5.38 用 5.12.2 节介绍的字符串比较函数和第 4 章介绍的数组排序技术编写一个程序, 按字母顺序列出字符串清单。用 10 个或 15 个城市名作为程序数据。

5.39 对图 5.29 的字符串复制和字符串连接函数编写两个版本, 一个用数组下标, 一个用指针与指针算法。

5.40 对图 5.29 的字符串比较函数编写两个版本, 一个用数组下标, 一个用指针与指针算法。

5.41 对图 5.29 的字符串 strlen 函数编写两个版本, 一个用数组下标, 一个用指针与指针算法。

### 特殊小节: 高级字符串操作练习

上述练习是本书的重点, 用于测试读者对基本字符串操作概念的理解。本节介绍一组中高级字符串操作练习, 这些题目的难度不一, 有的要花一两个小时来编写和实现, 有的要两三周来进行实验室讨论和实现, 有些是较难的小组项目。

5.42 (文本分析) 利用计算机的字符串操作功能可以用非常有趣的方法分析大作家的写作方法。许多人在分析莎士比亚是否真有其人, 有些学者认为, 许多重要证据表明, 莎士比亚实际上是 Christopher Marlowe 或其他作家的化名。研究人员通过计算机寻找这些作家在写作中的相似性。本练习介绍三种用计算机分析文章的方法。

a) 编写一个程序, 从键盘读取几行文本, 并打印一个表格, 显示文中字母的出现次数。例如, 下列短语:

```
To be, or not to be: that is the question:
```

包含一个 a、二个 b、不包含 c 等等。

- b) 编写一个程序，从键盘读取几行文本，并打印一个表格，显示文中单字符单词、双字符单词、三字符单词等的出现次数。例如，下列短语：

Whether 'tis nobler in the mind to suffer

包含：

| 字长 | 出现次数        |
|----|-------------|
| 1  | 0           |
| 2  | 2           |
| 3  | 2           |
| 4  | 2 (包括 'tis) |
| 5  | 0           |
| 6  | 2           |
| 7  | 1           |

- c) 编写一个程序，从键盘读取几行文本，并打印一个表格，显示文中每个单词的出现次数。程序的第一个版本应在表中按文中出现的顺序列出单词。例如，下列语句：

To be, or not to be: that is the question:  
Whether 'tis nobler in the mind to suffer

包含三个“to”、两个“be”、一个“or”等等。然后按字母顺序列出更有趣（也更有用）的打印输出。

- 5.43 (字处理) 字处理系统的一个重要功能是输入对齐，将单词与页面的左右边界对齐，从而产生漂亮的文档，就像是经过排版的，而不是直接输入的。计算机系统通过在行中的单词之间插入空格而让最右边的单词与右边界对齐。

编写一个程序，读取几行文本，并按对齐格式打印这些文本。假设文本在 8.5 英寸宽的纸上打印，左右页边留下 1 英寸的边距。假设计算机在水平方向每英寸打印 10 个字符。因此，可以在 6.5 英寸的空间打印 65 个字符。

- 5.44 (按不同格式打印日期) 可以按不同格式打印日期，两种常用格式如下所示：

07/21/55 和 July 21, 1955

编写一个程序，读取第一种日期格式，并打印第二种日期格式。

- 5.45 (支票保护) 计算机经常用于工资与账号支付应用等支票写入系统。许多怪事常常出现，如每周工资支票上错误地多写 1 百万美元。由于人为和机器的错误，使支票写入系统写出不正常的数值。系统设计人员在系统中建立控制，防止发出这种错误支票。

另一个严重问题是有些人故意改变支票金额，想窃取钱财。要防止改变支票金额，大多数支票写入系统采用支票保护 (check protection) 技术。

计算机打印的支票包含固定的可以打印金额的空格。假设支票中用八个空格写入每周工资，如果金额太大，则八个空格都会填满，例如：

1,230.60      (支票金额)  
12345678      (位置号)

另一方面，如果金额少于 1000 美元，则有些空格空着。例如：

```

  99.87
12345678

```

包含三个空格。如果打印支票时留下空格，则很容易篡改。要防止改变支票金额，许多支票写入系统插入如下星号：

```

***99.87
12345678

```

编写一个程序，输入支票上要打印的美元数，然后用支票保护格式打印金额，必要时加上星号，假设用九个空格打印金额。

- 5.46 (写出支票金额的大写) 继续上面的例子，设计支票写入系统以防止改变支票金额非常重要，一个常用的安全方法是写出支票金额的大写。即使支票的数字好改，大写金额也很难篡改。

许多计算机支票写入系统不写出支票金额的大写，也许主要原因是商用应用程序使用的大多数高级语言没有足够的字符串操作特性，另一个原因则是编写大写金额的程序比较复杂。

编写一个 C++ 程序，输入数字金额，输出大写金额。例如，金额 112.43 写成：

```
ONE HUNDRED TWELVE and 43/100
```

- 5.47 (莫尔斯码) 也许最著名的编码机制是莫尔斯码，是 1832 年由 Samuel Morse 创立的，用于电报系统使用。莫尔斯码对字母、数字和一些特殊符号（如圆点、逗号、分号）指定一系列点和线。在面向声音的系统中，点表示短音，线表示长音。点线表示还用于面向光的系统和面向信号标志系统。

单词之间用空格分开，没有点和线。在面向声音的系统中，空格表示为短时间不发声音。图 5.39 显示了莫尔斯码的国际化版本。

编写一个程序，读取一句英语短语，并将其编制成莫尔斯码，再用一个程序将莫尔斯码变成英语。莫尔斯码编码字母之间用一个空格，莫尔斯码编码单词之间用三个空格。

| 字符 | 代码    | 字符 | 代码      |
|----|-------|----|---------|
| A  | .-    | T  | -       |
| B  | -...  | U  | ..-     |
| C  | -.-.  | V  | ...-    |
| D  | -..   | W  | .-.     |
| E  | .     | X  | -..-    |
| F  | ..-   | Y  | -.-.-   |
| G  | --.   | Z  | ---.    |
| H  | ....  |    |         |
| I  | ..    | 数字 |         |
| J  | .---- | 1  | ....-   |
| K  | -.-   | 2  | ..---   |
| L  | .-..  | 3  | ...--   |
| M  | --    | 4  | ....-   |
| N  | -.    | 5  | .....   |
| O  | ---   | 6  | -.....  |
| P  | .-.-  | 7  | ---.... |
| Q  | ---.- | 8  | ----... |
| R  | .-.   | 9  | -----.  |
| S  | ...   | 0  | -----   |

图 5.39 莫尔斯码的国际化版本

- 5.48 (公制换算程序) 编写一个程序, 帮助用户进行公制换算。程序让用户指定单位名字符串 (如 centimeter、liter、gram 等表示公制, inch、quart、pound 等表示英制), 并回答下列简单问题:

```
"How many inches are in 2 meters?"  
"How many liters are in 10 quarts?"
```

程序应能认识无效转换。例如:

```
"How many feet in 5 kilograms?"
```

是无意义的, 因为 "feet" 是长度单位, 而 "kilograms" 是重量单位。

## 复杂字符串操作练习

- 5.49 (纵横填字谜产生器) 许多人都玩过纵横填字谜游戏, 但很少人建立过纵横填字谜。建立纵横填字谜是个复杂的问题, 这是个复杂字符串操作项目。程序员即使建立最简单的纵横填字谜, 也要解决大量问题。例如, 如何在计算机中表示纵横填字谜的网格? 是用一系列字符串, 还是用双下标数组? 程序员要提供程序直接引用的单词源 (即计算机化字典)。这些单词以怎样的形式存放来实现程序所需的复杂操作? 有的读者还想建立纵横填字谜的“线索”, 提示每行每列要打印的单词。仅仅打印一个空的纵横填字谜就不是一件简单的事。

## 第6章 类与数据抽象（一）

### 教学目标

- 了解封装与数据隐藏的软件工程概念
- 了解数据抽象和抽象数据类型（ADT）的符号
- 生成 C++ 的 ADT（即类）
- 了解如何生成、使用和删除类对象
- 控制对象数据成员和成员函数的访问
- 开始认识面向对象的价值

### 6.1 简介

下面开始介绍 C++ 中的面向对象。为什么把 C++ 中的面向对象推迟到第 6 章才开始介绍呢？原因是我们要建立的对象是由各个结构化程序组件构成，因此先要建立结构化程序的基础知识。

在第 1 章到第 5 章的“有关对象的思考”小节中，我们介绍了 C++ 中的面向对象编程的基本概念（即“对象思想”）和术语（即“对象语言”）。在这些“有关对象的思考”小节中，介绍了面向对象设计（object-oriented design, OOD）的方法：我们分析了典型问题的陈述，要求建立一个系统（电梯模拟程序），确定实现该系统所需的类，确定这些类对象的属性，确定这些类对象的行为，指定对象之间如何通过交互以完成系统的总体目标。

下面简要介绍面向对象的一些关键概念和术语。OOP 将数据（属性）和函数（行为）封装（encapsulate）到称为类（class）的软件包中，类的数据和成员是密切联系的。就像蓝图，建筑人员通过蓝图建造房子，而程序员则通过类生成对象。一个蓝图可以多次复用，建造多幢房子；一个类也可以多次复用，建立多个对象。类具有信息隐藏（information hiding）属性，即类对象只知道如何通过定义良好的接口（interface）与其他类对象通信，但通常不知道其他类的实现方法，实现细节隐藏在类中。我们可以熟练地开车，而不需要知道发动机、传递系统和燃油系统的工作原理。我们可以看到信息隐藏对良好的软件工程是多么重要。

在 C 语言和其他过程化编程语言（procedural programming language）中，编程是面向操作的（action-oriented），而在 C++ 中，编程是面向对象的（object-oriented）。在 C 语言中，编程的单位是函数（function），而在 C++ 中，编程的单位是类（class），对象最终要通过类实例化。

C 语言程序员的主要工作是编写函数，完成某个任务的一组操作构成函数，函数的组合则构成程序。数据在 C 语言中当然很重要，但这些数据只用于支持函数所要进行的操作。系统指定中的动词帮助 C 语言程序员确定一组用于实现系统的函数。

C++ 程序员把重点放在生成称为类的用户自定义类型（user-defined type），类也称为程序员定义类型（programmer-defined type）。每个类包含数据和操作数据的一组函数。类的数据部分称为数据成员（data member）。类的函数部分称为成员函数（member function，有些面向对象语言中也称

为方法)。int 等内部类型的实例称为变量 (variable)，而用户自定义类型 (即类) 的实例则称为对象 (object)。在 C++ 中，变量与对象常常互换使用，C++ 的重点是类而不是函数。系统指定中的名词帮助 C++ 程序员确定实现系统所需的用来生成对象的一组类。

C++ 中的类是由 C 语言中的 struct 演变而来的。介绍 C++ 类的开发之前，我们先使用结构建立用户自定义类型，通过介绍这种方法的缺点从而说明类的优点。

## 6.2 结构定义

结构是用其他类型的元素建立的聚合数据类型。考虑下列结构定义：

```
struct Time {  
    int hour;      // 0-23  
    int minute;    // 0-59  
    int second;    // 0-59  
};
```

结构定义用关键字 struct 引入。标识符 Time 是个结构标志 (structure tag)，命名结构定义并声明该结构类型 (structure type) 的变量。本例中，新类型名为 Time。结构定义花括号中声明的名称是结构的成员 (member)。同一结构的成员应有惟一名称，但两个不同结构可以包含同名成员而不会发生冲突。每个结构定义应以分号结尾。上述解释对后面要介绍的类也适用，C++ 中的结构和类是非常相似的。

Time 的定义包含三个 int 类型的成员 hour、minute 和 second。结构成员可以是任何类型，一个结构可以包含不同类型的成员。但是，结构不能包含自身的实例。例如，Time 类型的成员不能在 Time 的结构定义中声明，但该结构定义中可以包含另一 Time 结构的指针。当结构包含同一类型结构的指针时，称为自引用结构 (self-referential structure)。自引用结构用于形成链接数据结构，如链表、队列、堆栈和树等 (见第 15 章介绍)。

上述结构定义并没有在内存中保留任何空间，而是生成新的数据类型，用于声明变量。结构变量和其他类型的变量一样声明。下列声明：

```
Time timeObject, timeArray [ 10 ], *timePtr,  
    &timeRef = timeObject;
```

声明 timeObject 为 Time 类型变量，timeArray 为 10 个 Time 类型元素的数组，timePtr 为 Time 对象的指针，timeRef 为 Time 对象的引用 (用 timeObject 初始化)。

## 6.3 访问结构成员

访问结构成员或类成员时，使用成员访问运算符 (member access operator)，包括圆点运算符 (.) 和箭头运算符 (->)。圆点运算符通过对象的变量名或对象的引用访问结构和类成员。例如，要打印 timeObject 结构的 hour 成员，用下列语句：

```
cout << timeobject.hour;
```

要打印 timeRef 引用的结构的 hour 成员，用下列语句：



```
cout << timeRef.hour;
```

箭头运算符由负号(-)和大于号(>)组成,中间不能插空格,通过对象指针访问结构和类成员。假设指针timePtr声明为指向Time对象,结构timeObject的地址赋给timePtr。要打印指针为timePtr的timeObject结构的hour成员,用下列语句:

```
timePtr = &timeObject;
cout << timePtr->hour;
```

表达式timePtr->hour等价于(\*timePtr).hour,后者复引用指针并用圆点运算符访问hour成员。这里的括号是必需的,因为圆点运算符的优先级高于复引用指针运算符(\*)。箭头运算符和圆点运算符以及括号与方括号([])的优先级较高,仅次于第3章介绍的作用域运算符,结合律为从左向右。

#### 常见编程错误6.1

表达式(\*timePtr).hour指timePtr所指struct的hour成员。省略括号的\*timePtr.hour是个语法错误,因为“.”的优先级高于“\*”,表达式变成\*(timePtr.hour)。这是个语法错误,因为指针要用箭头运算符引用成员。

## 6.4 用 struct 实现用户自定义类型 Time

图6.1生成用户自定义类型Time,有三个整数成员hour、minute和second。程序定义一个Time类型的结构dinnerTime,并用圆点运算符初始化结构成员hour、minute和second的值分别为18、30和0。然后程序按军用格式(或所谓“通用格式”)和标准格式打印时间。注意打印函数接收常量Time结构的引用,从而通过引用将Time类型的结构传递给打印函数,避免了按值传入打印函数所涉及的复制开销,并用const防止打印函数修改Time结构。第7章将介绍const对象与const成员函数。

```
1 // Fig. 6.1: fig06_01.cpp
2 // Create a structure, set its members, and print it.
3 #include <iostream.h>
4
5 struct Time {      // structure definition
6     int hour;      // 0-23
7     int minute;    // 0-59
8     int second;    // 0-59
9 };
10
11 void printMilitary( const Time & ); // prototype
12 void printStandard( const Time & ); // prototype
13
14 int main()
15 {
16     Time dinnerTime;    // variable of new type Time
17
18     // set members to valid values
19     dinnerTime.hour = 18;
20     dinnerTime.minute = 30;
21     dinnerTime.second = 0;
22
23     cout << "Dinner will be held at ";
```

```
24  printMilitary( dinnerTime );
25  cout << " military time,\nwhich is ";
26  printStandard( dinnerTime );
27  cout << " standard time.\n";
28
29  // set members to invalid values
30  dinnerTime.hour = 29;
31  dinnerTime.minute = 73;
32
33  cout << "\nTime with invalid values: ";
34  printMilitary( dinnerTime );
35  cout << endl;
36  return 0;
37 }
38
39 // Print the time in military format
40 void printMilitary( const Time &t )
41 {
42     cout << ( t.hour < 10 ? "0" : "" ) << t.hour << ":"
43           << ( t.minute < 10 ? "0" : "" ) << t.minute;
44 }
45
46 // Print the time in standard format
47 void printStandard( const Time &t )
48 {
49     cout << ( ( t.hour == 0 || t.hour == 12 ) ?
50             12 : t.hour % 12 )
51           << ":" << ( t.minute < 10 ? "0" : "" ) << t.minute
52           << ":" << ( t.second < 10 ? "0" : "" ) << t.second
53           << ( t.hour < 12 ? " AM" : " PM" );
54 }
```

**输出结果:**

Dinner will be held at 18:30 military time,  
which is 6:30:00 PM standard time.

Time with invalid values: 29:73

图 6.1 生成结构、设置结构成员和打印该结构

**性能提示 6.1**

结构通常按值调用传递。要避免复制结构的开销，可以按引用调用传递结构。

**软件工程视点 6.1**

要避免按值调用传递的开销而且保护调用者的原始数据不被修改，可以将长度很大的参数作为 const 引用传递。

用这种方式通过结构生成新数据类型有一定的缺点。由于初始化并不是必须的，因此就可能出现未初始化的数据，从而造成不良后果。即使数据已经初始化，也可能没有正确地初始化。因为程序能够直接访问数据，所以无效数据可能赋给结构成员（如图 6.1）。在第 30 行和第 31 行，程序很容易向 Time 对象 dinnerTime 的 hour 和 minute 成员传递错值。如果 struct 的实现方法改变（例如时间可以表示为从午夜算起的秒数），则所有使用这个 struct 的程序都要改变。这是因为程序员直接操作数据类型。没有一个“接口”保证程序员正确使用数据类型并保持数据的一致状态。

### 软件工程视点 6.2

一定要编写易于理解和易于维护的程序。不断改变是规则而不是例外。程序员应预料到代码要经常改变。可以看出，类能够提高程序的可修改性。

还有其他与C语言式结构相关的问题。在C语言中，结构不能作为一个单位打印，而要一次一个地打印和格式化结构成员。可以编写一个函数，以某种格式打印结构成员。第8章“运算符重载”中演示了如何重载<<运算符，使结构类型或类类型的对象能够方便地打印。在C语言中，结构不能整体进行比较，而只能一个成员一个成员地比较。第8章还会演示如何重载相等运算符与关系运算符，比较C++结构类型或类类型的对象。

下一节重新将Time结构实现为C++类，并演示用类生成抽象数据类型（abstract data type）的好处。从中将会看到，C++中类和结构的用法基本相同，差别在于各自的成员相关的默认访问能力不同。

## 6.5 用类实现Time抽象数据类型

类使程序员可以构造对象的属性（attribute，表示为数据成员）和行为（behavior）或操作（operation，表示为成员函数）。C++中用关键字class定义包含数据成员和成员函数的类型。

成员函数在有些面向对象编程语言中也称为方法（method），响应对象接收的消息（message）。消息对应于一个对象发给另一个对象或由函数发给对象的成员函数调用。

一旦定义了一个类，可以用类名声明该类的对象。图6.2显示了Time类的简单定义。

Time类定义以关键字class开始。类定义体放在左右花括号（{}）之间，类定义用分号终止。Time类定义和Time类结构定义各包含三个整型成员hour、minute和second。

```
1 class Time {
2 public:
3     Time();
4     void setTime( int, int, int);
5     void printMilitary();
6     void printStandard();
7 private:
8     int hour;           // 0-23
9     int minute;         // 0-59
10    int second;         // 0-59
11 };
```

图 6.2 Time 类的简单定义

### 常见编程错误 6.2

忘记类或结构定义结束时的分号是个语法错误。

类定义的其他部分是新内容。public:和private:标号称为成员访问说明符（member access specifier）。在程序能访问Time类对象的任何地方都可以访问任何在成员访问说明符public后面（和下一个成员访问说明符之前）声明的数据成员和成员函数。成员访问说明符private后面（和下一个成员访问说明符之前）声明的数据成员和成员函数只能由该类的成员函数访问。成员访问说明符总是加上冒号，可以在类定义中按任何顺序多次出现。本文余下部分使用不带冒号的成员访问说明符public和private。第9章还将介绍另一个成员访问说明符protected，并介绍继承及其在面向对象编程中的作用。

## 编程技巧 6.1

每个成员访问说明符只在类定义中使用一次，这样可以增加清晰性与可读性。将 public 成员放在前面，便于寻找。

类定义中的访问说明符 public 后面是成员函数 Time、setTime、printMilitary 和 printStandard 的函数原型。这些函数是类的 public 成员函数（或 public 服务、public 行为、类的接口）。类的客户（client，即程序中的用户部分）使用这些函数操作该类的数据。

注意与类名相同的成员函数，称为该类的构造函数（constructor）。构造函数是个特殊成员函数，该函数初始化类对象的数据成员。类的构造函数在生成这个类的对象时自动调用。一个类常常有几个构造函数，这是通过函数重载完成的。注意，构造函数不指定返回类型。

## 常见编程错误 6.3

对构造函数指定返回类型或返回值是个语法错误。

成员访问说明符 private 后面有三个整型成员，表示类的这些数据成员只能让成员函数访问（下一章会介绍还可由类的友元访问）。这样，数据成员只能由类定义中出现函数原型的 4 个函数（和类的友元）访问。数据成员通常放在类的 private 部分，成员函数通常放在 public 部分。稍后会介绍，也可以用 private 成员函数和 public 数据，但比较少见，这不是好的编程习惯。

定义类之后，可以在声明中将其当作类型，如下所示：

```
Time sunset,                // object of type Time
    arrayOfTimes[ 5 ],      // array of Time objects
    *pointerToTime,         // pointer to a Time object
    &dinnerTime = sunset;    // reference to a Time object
```

类名成为新的类型说明符。一个类可以有多个对象，就像 int 类型的变量可以有多个。程序员可以在需要时生成新的类类型，因此 C++ 是个可扩展语言（extensible language）。

图 6.3 使用 Time 类。程序实例化 Time 类的一个对象 t。当对象实例化时，Time 构造函数自动调用，显式地将每个 private 数据成员初始化为 0。然后按军用格式和标准格式打印时间，确保成员已经正确地初始化。然后用 setTime 成员函数设置时间，并再次按两种格式打印时间。接着用 setTime 成员函数设置时间为无效值，并再次按两种格式打印时间。

```
1 // Fig. 6.3: fig06_03.cpp
2 // Time class.
3 #include <iostream.h>
4
5 // Time abstract data type (ADT) definition
6 class Time {
7 public:
8     Time();                // constructor
9     void setTime( int, int, int ); // set hour, minute, second
10    void printMilitary();    // print military time format
11    void printStandard();    // print standard time format
12 private:
13    int hour;               // 0 - 23
14    int minute;             // 0 - 59
15    int second;             // 0 - 59
16 };
17
```

```
18 // Time constructor initializes each data member to zero.
19 // Ensures all Time objects start in a consistent state.
20 Time::Time() { hour = minute = second = 0; }
21
22 // Set a new Time value using military time. Perform validity
23 // checks on the data values. Set invalid values to zero.
24 void Time::setTime( int h, int m, int s )
25 {
26     hour = ( h >= 0 && h < 24 ) ? h : 0;
27     minute = ( m >= 0 && m < 60 ) ? m : 0;
28     second = ( s >= 0 && s < 60 ) ? s : 0;
29 }
30
31 // Print Time in military format
32 void Time::printMilitary()
33 {
34     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
35           << ( minute < 10 ? "0" : "" ) << minute;
36 }
37
38 // Print Time in standard format
39 void Time::printStandard()
40 {
41     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
42           << ":" << ( minute < 10 ? "0" : "" ) << minute
43           << ":" << ( second < 10 ? "0" : "" ) << second
44           << ( hour < 12 ? " AM" : " PM" );
45 }
46
47 // Driver to test simple class Time
48 int main()
49 {
50     Time t; // instantiate object t of class Time
51
52     cout << "The initial military time is ";
53     t.printMilitary();
54     cout << "\nThe initial standard time is ";
55     t.printStandard();
56
57     t.setTime( 13, 27, 6 );
58     cout << "\n\nMilitary time after setTime is ";
59     t.printMilitary();
60     cout << "\nStandard time after setTime is ";
61     t.printStandard();
62
63     t.setTime( 99, 99, 99 ); // attempt invalid settings
64     cout << "\n\nAfter attempting invalid settings:"
65           << "\nMilitary time: ";
66     t.printMilitary();
67     cout << "\nStandard time: ";
68     t.printStandard();
69     cout << endl;
70     return 0;
71 }
```

**输出结果:**

```
The initial military time is 00::00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00::00
Standard time: 12:00:00 AM
```

图 6.3 用类实现抽象数据类型 Time

注意数据成员 `hour`、`minute` 和 `second` 前面使用成员访问说明符 `private`。类的 `private` 数据成员通常只能在类中访问（下一章会介绍，还可由类的友元访问）。从本例中可以看出，类的客户不关心类中的实际数据表达。例如，类完全可以用从午夜算起的秒数表示时间，这时客户可以用相同的 `public` 成员函数取得相同的结果而并不注意类中的变化。从这种意义上说，类的实现是向客户隐藏起来的。这种信息隐藏提高了程序的可修改性，简化了客户对类的理解。

**软件工程视点 6.3**

类的客户使用类时不必知道类的内部实现细节。如果类的内部实现细节改变（例如为了提高性能），只要类的接口保持不变，类的客户源代码就不必改变（但客户可能需要重新编译），这样就更容易修改系统。

在这个程序中，`Time` 构造函数只是将数据成员初始化为 0（即上午 12 时的军用时间格式），因此就保证对象生成时具有一致状态。`Time` 对象的数据成员中不可能保存无效值，因为生成 `Time` 对象时自动调用构造函数，后面客户对数据成员的修改都是由 `setTime` 函数完成的。

**软件工程视点 6.4**

成员函数通常比非面向对象编程中的函数更短，因为数据成员中存放的数据已经由构造函数和保存新数据的成员函数验证。由于数据已经是对象，成员函数调用通常没有参数或比非面向对象语言中调用的典型函数的参数更少。这样，调用简化了，函数定义简化了，函数原型也简化了。

注意，类的数据成员无法在类体中声明时初始化，而要用类的构造函数初始化，也可以用给它们设值的函数赋值。

**常见编程错误 6.4**

想在类定义中显式地将类的数据成员初始化是个语法错误。

与类同名而前面加上波浪号（~）的函数称为类的析构函数（`destructor`）（本例没有显式地加上析构函数，系统会插入一个析构函数）。析构函数在系统收回对象的内存之前对每个类对象进行清理工作。析构函数不带参数，无法重载。本章稍后和第 7 章将详细介绍构造函数与析构函数。

注意，类向外部提供的函数要加上 `public` 标号。`public` 函数实现类向客户提供的行为或服务，通常称为类的接口或 `public` 接口。

**软件工程视点 6.5**

客户能访问类的接口，但不能访问类的实现方法。

类定义包含类的数据成员和成员函数的声明。成员函数的声明就是本书前面介绍的函数原型。成员函数可以在类的内部定义，但在类的外部定义函数是个良好的习惯。

**软件工程视点 6.6**

在类定义中（通过函数原型）声明成员函数而在类定义外定义这些成员函数，可以区分类的接口与实现方法。这样可以实现良好的软件工程，类的客户不能看到类成员函数的实现方法。

注意图 6.3 类定义中每个成员函数定义使用的二元作用域运算符 (::)。定义类和声明成员函数后，就要定义成员函数。类的每个成员函数可以直接在类定义体中定义（而不是包括类的函数原型），也可以在类定义体之后定义成员函数。在类定义体之后定义成员函数时，函数名前面要加上类名和二元作用域运算符 (::)。由于不同类可能有同名成员，因此要用二元作用域运算符将成员名与类名联系起来，惟一标识某个类的成员函数。

**常见编程错误 6.5**

在类的外部定义成员函数时，省略函数名中的类名和二元作用域运算符是个语法错误。

尽管类定义中声明的成员函数可以在类定义之外定义，但成员函数仍然在类范围（class's scope）中，即只有该类的其他成员知道它的名称，除非通过类对象、引用类对象或类对象指针进行引用。稍后将详细介绍类范围。

如果在类定义中定义成员函数，则该成员函数自动成为内联函数。在类体之后定义成员函数时，可以用关键字 inline 指定其为内联函数。记住，编译器有权不把内联函数放进程序块中。

**性能提示 6.2**

在类定义内定义小的成员函数将自动使该函数成为内联函数（如果编译器选择这么做），这样虽然可以提高性能，但不能提高软件工程质量，因为类的客户能看到函数实现方法。

**软件工程视点 6.7**

只有最简单的成员函数才能在类的首部中定义。

有趣的是 printMilitary 和 printStandard 成员函数没有参数。这是因为成员函数隐式知道对调用的特定 Time 对象打印数据成员。这样就使成员函数调用比过程式编程中的传统函数调用更为简练。

**测试与调试提示 6.1**

成员函数调用通常不带参数或比非面向对象语言中的传统函数调用参数少得多，从而减少传递错误参数、错误参数类型或错误参数个数的机会。

**软件工程视点 6.8**

利用面向对象编程方法通常能减少传递的参数个数，从而简化函数调用。这个面向对象编程好处是由于在对象中封装数据成员和成员函数之后，成员函数有权访问数据成员。

类能简化编程，因为客户（或类对象用户）仅需关心对象中封装或嵌入的操作。这种操作通常是面向客户的，而不是面向实现方法的。客户不必关心类的实现方法（当然客户需要正确和有效的实现方法）。接口不是没有改变，只是不像实现方法那样经常改变而已。实现方法改变时，与实现方法有关的代码也要相应改变。通过隐藏实现方法，可以消除程序中与实现方法有关的代码。

**软件工程视点 6.9**

本书的中心主题是“复用、复用、再复用”。我们将认真介绍几个提高复用性的技术，着重介绍“建立宝贵的类”和建立宝贵的“软件资产”。

类通常不需要从头生成,可以从其他提供新类可用的属性和行为的类派生而来,类中可以包括其他类对象作为成员。这种软件复用可以大大提高程序员的工作效率。从现有类派生新类称为继承 (inheritance),将在第9章介绍。把其他类对象作为类的成员称为复合 (composition),将在第7章介绍。

不熟悉面向对象编程的人常常担心对象会很大,因为它们要包含数据和函数。逻辑上的确如此,程序员可以把对象看成要包含数据和函数,但实际中并不是这样。

### 性能提示 6.3

实际对象只包含数据,因此要比包含函数的对象小得多。对类名或该类的对象采用 sizeof 运算符时,只得到该类的数据长度。编译器生成独立于所有类对象的类成员函数副本(只有一份)。当然,因为每个对象的数据是不同的,所以每个对象需要自己的类数据副本。该函数代码是不变的(或称为可重入码或纯过程),因此可以在一个类的所有对象之间的共享。

## 6.6 类范围与访问类成员

类的数据成员(类定义中声明的变量)和成员函数(类定义中声明的函数)属于该类的类范围(class's scope)。非成员函数在文件范围(file scope)中定义。

在类范围中,类成员可由该类的所有成员函数直接访问,也可以用名称引用。在类范围外,类成员是通过一个对象的句柄引用,可以是对象名、对象引用或对象指针(第7章将介绍,每次引用对象中的数据成员和成员函数时,编译器插入一个隐式句柄)。

类的成员函数可以重载,但只能由这个类的其他成员函数重载。要重载成员函数,只要在类定义中提供该重载函数每个版本的原型,并对该重载函数每个版本提供不同的函数定义。

成员函数在类中有函数范围(function scope),成员函数内定义的变量只能在该函数内访问。如果成员函数定义与类范围内的变量同名的变量,则在函数范围内,函数范围内的变量掩盖类范围内的变量。这种隐藏变量可以通过在前面加上类名和作用域运算符(::)而访问。隐藏的全局变量可以用一元作用域运算符访问(见第3章)。

访问类成员的运算符与访问结构成员的运算符是相同的。圆点成员选择运算符(.)与对象名或对象引用组合,用于访问对象成员。箭头成员选择运算符(->)与对象指针组合,用于访问对象成员。

图6.4的程序用简单的Count类和public数据成员x(int类型)以及public成员函数print演示如何用成员选择运算符访问类成员。程序实例化三个Count类型的变量——counter、counterRef(Count对象的引用)和counterPtr(Count对象的指针)。变量counterRef定义为引用counter,变量counterPtr定义为指向counter。注意,这里将数据成员x设置为public,只是为了演示public成员利用句柄(如名称、引用或指针)即可访问。前面曾介绍过,数据通常指定为private,第9章“继承”中将介绍有时可以将数据指定为protected。

```
1 // Fig. 6.4: fig06_04.cpp
2 // Demonstrating the class member access operators . and ->
3 //
4 // CAUTION: IN FUTURE EXAMPLES WE AVOID PUBLIC DATA!
5 #include <iostream.h>
6
7 // Simple class Count
```



```
8 class Count {
9 public:
10     int x;
11     void print() { cout << x << endl; }
12 };
13
14 int main()
15 {
16     Count counter,           // create counter object
17         *counterPtr = &counter, // pointer to counter
18         &counterRef = counter; // reference to counter
19
20     cout << "Assign 7 to x and print using the object's name: ";
21     counter.x = 7;           // assign 7 to data member x
22     counter.print();         // call member function print
23
24     cout << "Assign 8 to x and print using a reference: ";
25     counterRef.x = 8;        // assign 8 to data member x
26     counterRef.print();      // call member function print
27
28     cout << "Assign 10 to x and print using a pointer: ";
29     counterPtr->x = 10;       // assign 10 to data member x
30     counterPtr->print();      // call member function print
31     return 0;
32 }
```

**输出结果：**

```
Assign 7 to x and print using the object's name: 7
Assign 8 to x and print using a reference: 8
Assign 10 to x and pring using a pointer: 10
```

图 6.4 通过各种句柄访问对象的数据成员和成员函数

## 6.7 接口与实现方法的分离

良好软件工程的一个基本原则是将接口与实现方法分离,这样可以更容易修改程序。就类的客户而言,类实现方法的改变并不影响客户,只要类的接口保持不变即可(类的功能可能扩展到原接口以外)。

**软件工程视点 6.10**

将类声明放在使用该类的任何客户的头文件中,这就形成类的 public 接口(并向客户提供调用类成员函数所需的函数原型)。将类成员函数的定义放在源文件中,这就形成类的实现方法。

**软件工程视点 6.11**

类的客户使用类时不需要访问类的源代码,但客户需要连接类的目标码。这样就可以由独立软件供应商(ISV)提供类库进行销售和发放许可证。ISV 只在产品中提供头文件和目标模块,不提供专属信息(例如源代码)。C++ 用户可以享用更多的 ISV 生产的类库。

实际上,任何事情都不是十全十美的。头文件中包含一些实现部分,并隐藏了实现方法的其他部分。例如,内联成员函数要放在头文件中,使编译器编译客户代码时,该客户代码能够包含内联

函数定义。`private`成员列在头文件的类定义中，因此客户虽然无法访问`private`成员，但能看到这些成员。第7章将介绍如何用代理类从类的客户中隐藏类的`private`数据。

#### 软件工程视点 6.12

对类接口很重要的信息应放在头文件中。只在类内部使用而类的客户不需要的信息应放在不发表的源文件中。这是最低权限原则的又一个例子。

图 6.5 将图 6.3 的程序分解为多个文件。建立 C++ 程序时，每个类定义通常放在头文件中，类的成员函数定义放在相同基本名字的源代码文件（source-code file）中。在使用类的每个文件中包含头文件（通过 `#include`），而源代码文件编译并连接包含主程序的文件。编译器文档中介绍了如何编译和连接由多个源文件组成的程序。

图 6.5 包含声明 `Time` 类的 `timel.h` 头文件、定义 `Time` 类成员函数的 `timel.cpp` 文件和定义 `main` 函数的 `fig06_05.cpp` 文件。这个程序的输出与图 6.3 的输出相同。

```

1 // Fig. 6.5: timel.h
2 // Declaration of the Time class.
3 // Member functions are defined in timel.cpp
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME1_H
7 #define TIME1_H
8
9 // Time abstract data type definition
10 class Time {
11 public:
12     Time(); // constructor
13     void setTime( int, int, int ); // set hour, minute, second
14     void printMilitary(); // print military time format
15     void printStandard(); // print standard time format
16 private:
17     int hour; // 0 - 23
18     int minute; // 0 - 59
19     int second; // 0 - 59
20 };
21
22 #endif
23 // Fig. 6.5: timel.cpp
24 // Member function definitions for Time class.
25 #include <iostream.h>
26 #include "timel.h"
27
28 // Time constructor initializes each data member to zero.
29 // Ensures all Time objects start in a consistent state.
30 Time::Time() { hour = minute = second = 0; }
31
32 // Set a new Time value using military time. Perform validity
33 // checks on the data values. Set invalid values to zero.
34 void Time::setTime( int h, int m, int s )
35 {
36     hour = ( h >= 0 && h < 24 ) ? h : 0;
37     minute = ( m >= 0 && m < 60 ) ? m : 0;
38     second = ( s >= 0 && s < 60 ) ? s : 0;
39 }

```

```

40
41 // Print Time in military format
42 void Time::printMilitary()
43 {
44     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
45         << ( minute < 10 ? "0" : "" ) << minute;
46 }
47
48 // Print time in standard format
49 void Time::printStandard()
50 {
51     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
52         << ":" << ( minute < 10 ? "0" : "" ) << minute
53         << ":" << ( second < 10 ? "0" : "" ) << second
54         << ( hour < 12 ? " AM" : " PM" );
55 }
56 // Fig. 6.5: fig06_05.cpp
57 // Driver for Time1 class
58 // NOTE: Compile with timel.cpp
59 #include <iostream.h>
60 #include "timel.h"
61
62 // Driver to test simple class Time
63 int main()
64 {
65     Time t;    // instantiate object t of class time
66
67     cout << "The initial military time is";
68     t.printMilitary();
69     cout << "\nThe initial standard time is";
70     t.printStandard();
71
72     t.setTime( 13, 27, 6 );
73     cout << "\n\nMilitary time after setTime is";
74     t.printMilitary();
75     cout << "\nStandard time after setTime is";
76     t.printStandard();
77
78     t.setTime( 99, 99, 99 ); // attempt invalid settings
79     cout << "\n\nAfter attempting invalid settings:\n"
80         << "Military time:";
81     t.printMilitary();
82     cout << "\nStandard time:";
83     t.printStandard();
84     cout << endl;
85     return 0;
86 }

```

**输出结果:**

```

The initial military time is 00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:

```

```
Military time: 00:00
Standard time: 12:00:00 AM
```

图 6.5 将 Time 类的接口与实现方法分离

注意类声明放在下列预处理代码中：

```
// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H
    ...
#endif
```

建立大程序时，其他定义和声明也放在头文件中。上述预处理指令使得在定义了TIME1\_H名字时不再包含#ifndef和#endif之间的代码。如果文件中原先没有包含头文件，则TIME1\_H名字由#define指令定义，并使该文件包含头文件语句。如果文件中已经包含头文件，则TIME1\_H名字已经定义，不再包含头文件语句。多次包含头文件语句通常发生在大程序中，许多头文件本身已经包含其他头文件。注意：预处理指令中符号化常量名使用的规则是把头文件名中圆点(.)换成下划线。

#### 测试与调试提示 6.2

用#ifndef、#define和#endif预处理指令防止一个程序中多次包含相同的头文件。

#### 编程技巧 6.2

头文件的#ifndef和#define预处理指令中用头文件名，并将圆点换成下划线。

## 6.8 控制对成员的访问

成员访问说明符public和private（和第9章“继承”中介绍的protected）可以控制类数据成员和成员函数的访问。类的默认访问模式是private，因此类的首部和第一个标号之间的所有成员的类型都是private。每个标号之后，采用该标号表示的方式，直到遇到下一个标号或遇到类定义的右花括号}。标号public、private和protected可以重复，但这种情况不常用，容易造成混乱。

类的private成员只能由类的成员函数（和第7章介绍的友元）访问，public成员则可以由程序中的任何函数访问。

public成员的主要用途是向类的客户提供类的服务（行为），这组服务形成类的public接口。类的客户不必关心类如何完成任务。类的private成员和public成员函数的定义是类的客户无法访问的。这些组件形成类的实现方法（implementation）。

#### 软件工程视点 6.13

C++提倡程序独立于实现方法。对于独立于实现方法的代码，改变类的实现方法时，代码不需要改变，但可能需要重新编译。

#### 常见编程错误 6.6

除了由类的成员函数（和第7章介绍的友元）访问外，其他函数想访问类的private成员是个语法错误。

图 6.6 演示了private类成员只能用public成员函数通过public类接口访问。编译这个程序时，编译器产生两个错误，表示每个语句中指定的private成员无法访问。图 6.6 包含time1.h并和图 6.5 的time1.cpp一起编译。

**编程技巧 6.3**

如果在类定义中先列出 private 成员，尽管程序默认的访问模式为 private，但最好还是显式使用 private 标号，这样可以使程序更清晰。我们喜欢先列出 public 成员以强调类的接口。

```
1 // Fig. 6.6: fig06_06.cpp
2 // Demonstrate errors resulting from attempts
3 // to access private class members.
4 #include <iostream.h>
5 #include "time.h"
6
7 int main()
8 {
9     Time t;
10
11     // Error: 'Time::hour' is not accessible
12     t.hour = 7;
13
14     // Error: 'Time::minute' is not accessible
15     cout << "minute = " << t.minute;
16
17     return 0;
18 }
```

**输出结果：**

```
Compiling FIG06_06.CPP:
Error FIG06_06.CPP 12: 'Time::hour' is not accessible
Error FIG06_06.CPP 15: 'Time::minute' is not accessible
```

图 6.6 访问类的 private 成员的错误

**编程技巧 6.4**

尽管 public 和 private 标号可以重复和混合，但最好先将所有 public 成员列成一组，然后将所有 private 成员列成一组，这样可以使客户集中注意类的 public 接口，而不是注意类的实现方法。

**软件工程视点 6.14**

让类的所有数据成员保持 private。让 public 成员函数设置 private 数据成员的值并取得 private 数据成员的值。这种结构能隐藏类的实现方法，减少错误和提高程序的可修改性。

类的客户可能是另一类的成员函数，也可能是全局函数（即文件中类 C 语言的“松散”函数，不是任何类的成员函数）。

类成员的默认访问方式为 private。类成员的访问方式可以显式设置为 public、protected（见第 9 章）和 private。struct 成员的默认访问方式为 public。struct 的成员的访问方式也可以设置为 public、protected 或 private。

**软件工程视点 6.15**

类设计人员用 public、protected 或 private 成员实现信息隐藏和最低权限原则。

类数据为 private 并不表示客户不能改变这个数据。客户可以通过这个类的成员函数或友元改变这个数据，但这些函数的设计应保证数据完整性。

访问类的 private 数据应当用称为访问函数（access function）或访问方法（accessor method）的成员函数严格控制。例如，要让客户读取 private 数据的值，类可以提供一個 get 函数。要让客户修

改 private 数据的值, 类可以提供一个 set 函数。这种修改似乎会破坏 private 数据的专用性, 但 set 成员函数可以提供数据验证功能 (如范围检查), 保证数值设置正确, set 函数也可以在接口使用的数字形式与实现方法使用的数据形式之间进行换算。get 函数不必以原始形式显示数据, 该函数可以编辑数据, 限制客户可以看到的数字。

#### 软件工程视点 6.16

类设计人员不必提供每个 private 数据成员的 get 和 set 函数, 只在需要时才提供数据成员的 get 和 set 函数。

#### 测试与调试提示 6.3

将类的数据成员指定为 private、类的成员函数指定为 public 有助于调试, 因为数据操作问题局部化在类成员函数或类的友元中。

## 6.9 访问函数与工具函数

并非所有成员函数都要用 public 指定为类接口的一部分。有些成员函数保持 private, 作为类中其他函数的工具函数 (utility function)。

#### 软件工程视点 6.17

成员函数分为几大类: 读取和返回私有数据成员值的函数、设置私有数据成员值的函数、实现类特性的函数和进行各种操作的函数 (如初始化类对象、指定类对象、将类与内部类型或其他类进行相互转换以及处理类对象内存)。

访问函数可以读取和显示数据。访问函数的另一常见用法是测试条件的真假, 这种函数称为判定函数 (predicate function)。任何容器类都有的 isEmpty 函数 (如链表、堆栈和队列) 就是判定函数。程序先测试 isEmpty, 再从容器对象中读取下一个项目。判定函数 isFull 用于测试容器类对象还有没有多余的存储空间。Time 类的判定函数包括 isAM 和 isPM。

图 6.7 演示了工具函数 (或称为帮助函数) 的使用。工具函数不是类接口的一部分, 而是 private 成员函数, 支持类中其他函数的操作。类的客户不能使用工具函数。

```

1 // Fig. 6.7: salesp.h
2 // SalesPerson class definition
3 // Member functions defined in salesp.cpp
4 #ifndef SALESP_H
5 #define SALESP_H
6
7 class SalesPerson {
8 public:
9     SalesPerson();           // constructor
10    void getSalesFromUser(); // get sales figures from keyboard
11    void setSales( int, double ); // User supplies one month's
12                                   // sales figures.
13    void printAnnualSales();
14
15 private:
16    double totalAnnualSales(); // utility function
17    double sales[ 12 ];       // 12 monthly sales figures
18 };
19
20 #endif

```

```
21 // Fig. 6.7: salesp.cpp
22 // Member functions for class SalesPerson
23 #include <iostream.h>
24 #include <iomanip.h>
25 #include "salesp.h"
26
27 // Constructor function initializes array
28 SalesPerson::SalesPerson()
29 {
30     for ( int i = 0; i < 12; i++ )
31         sales[ i ] = 0.0;
32 }
33
34 // Function to get 12 sales figures from the user
35 // at the keyboard
36 void SalesPerson::getSalesFromUser()
37 {
38     double salesFigure;
39
40     for ( int i = 0; i < 12; i++ ) {
41         cout << "Enter sales amount for month "
42              << i + 1 << ": ";
43         cin >> salesFigure;
44         setSales( i, salesFigure );
45     }
46 }
47
48 // Function to set one of the 12 monthly sales figures.
49 // Note that the month value must be from 0 to 11.
50 void SalesPerson::setSales( int month, double amount )
51 {
52     if ( month >= 0 && month < 12 && amount > 0 )
53         sales[ month ] = amount;
54     else
55         cout << "Invalid month or sales figure" << endl;
56 }
57
58 // Print the total annual sales
59 void SalesPerson::printAnnualSales()
60 {
61     cout << setprecision( 2 )
62          << setiosflags( ios::fixed | ios::showpoint )
63          << "\nThe total annual sales are: $"
64          << totalAnnualSales() << endl;
65 }
66
67 // Private utility function to total annual sales
68 double SalesPerson::totalAnnualSales()
69 {
70     double total = 0.0;
71
72     for ( int i = 0; i < 12; i++ )
73         total += sales[ i ];
74
75     return total;
```

```
76 }
77 // Fig. 6.7: fig06_07.cpp
78 // Demonstrating a utility function
79 // Compile with salesp.cpp
80 #include "salesp.h"
81
82 int main()
83 {
84     SalesPerson s;           // create SalesPerson object s
85
86     s.getSalesFromUser();    // note simple sequential code
87     s.printAnnualSales();    // no control structures in main
88     return 0;
89 }
```

**输出结果:**

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 2024.97
Enter sales amount for month 12: 5923.92
```

The total annual sales are: \$60120.58

图 6.7 使用工具函数

`SalesPerson`类中的表示12个月销售数据的数组用构造函数初始化为0,并用`setSales`函数设置为用户提供的值。`public`成员函数`printAnnualSales`打印最近12个月的总销售额。工具函数`totalAnnualSales`为`printAnnualSales`计算12个月的总销售额。成员函数`printAnnualSales`将销售数据转换为美元金额格式。

注意`main`中只有一个简单的成员函数调用,没有任何控制结构。

**软件工程视点 6.18**

面向对象编程的一个现象是定义类之后,生成和操作这个类的对象通常只要一个简单的成员函数调用,没有任何或只有少量控制结构。相反,类成员函数的实现则通常需要控制结构。

## 6.10 初始化类对象:构造函数

生成类对象时,其成员可以用类的构造函数初始化。构造函数是与类同名的成员函数。程序员提供的构造函数在每次生成类对象(实例化)时自动调用。构造函数可以重载,提供初始化类对象的不同方法。数据成员应在类的构造函数中初始化或在生成对象之后设置其数值。

**常见编程错误 6.7**

类的数据成员不能在类定义中初始化。



**常见编程错误 6.8**

试图声明构造函数的返回类型和返回值是个语法错误。

**编程技巧 6.5**

适当时候（通常都是）应提供一个构造函数，保证每个对象正确地初始化为有意义的值。特别是指针数据类型应初始化为合法指针值或0。

**测试与调试提示 6.4**

每个修改对象的 private 数据成员的成员函数（和友元）应确保该数据保持一致状态。

声明类对象时，可以在括号中提供初始化值，放在对象名后面和分号前面。这些初始化值作为参数传递给类的构造函数。稍后会举几个构造函数调用（constructor call）的例子（注意：尽管程序员不显式调用构造函数，但程序员仍然可以提供数据，作为参数传递给构造函数）。

## 6.11 在构造函数中使用默认参数

图 6.5 time1.cpp 中的构造函数将 hour、minute 和 second 初始化为 0（即军用时间午夜 12 时）。构造函数可以包含默认参数。图 6.8 重新定义 Time 的构造函数，该函数中每个变量的默认参数为 0。通过提供构造函数默认参数，即使在构造函数调用中不提供数值，对象也能利用默认参数初始化为一致状态。程序员提供所有参数默认值（或显式不要求参数）的构造函数也称为默认构造函数（default constructor），即可以不用参数而调用的构造函数。一个类只能有一个默认构造函数。

```
1 // Fig. 6.8: time2.h
2 // Declaration of the Time class.
3 // Member functions are defined in time2.cpp
4
5 // preprocessor directives that
6 // prevent multiple inclusions of header file
7 #ifndef TIME2_H
8 #define TIME2_H
9
10 // Time abstract data type definition
11 class Time {
12 public:
13     Time( int = 0, int = 0, int = 0 ); // default constructor
14     void setTime( int, int, int ); // set hour, minute, second
15     void printMilitary();           // print military time format
16     void printStandard();           // print standard time format
17 private:
18     int hour;           // 0 - 23
19     int minute;         // 0 - 59
20     int second;         // 0 - 59
21 };
22
23 #endif
24 // Fig. 6.8: time2.cpp
25 // Member function definitions for Time class.
26 #include <iostream.h>
27 #include "time2.h"
28
```

```

29 // Time constructor initializes each data member to zero.
30 // Ensures all Time objects start in a consistent state.
31 Time::Time( int hr, int min, int sec )
32 { setTime( hr, min, sec ); }
33
34 // Set a new Time value using military time. Perform validity
35 // checks on the data values. Set invalid values to zero.
36 void Time::setTime( int h, int m, int s )
37 {
38     hour   = ( h >= 0 && h < 24 ) ? h : 0;
39     minute = ( m >= 0 && m < 60 ) ? m : 0;
40     second = ( s >= 0 && s < 60 ) ? s : 0;
41 }
42
43 // Print Time in military format
44 void Time::printMilitary()
45 {
46     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
47           << ( minute < 10 ? "0" : "" ) << minute;
48 }
49
50 // Print Time in standard format
51 void Time::printStandard()
52 {
53     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
54           << ":" << ( minute < 10 ? "0" : "" ) << minute
55           << ":" << ( second < 10 ? "0" : "" ) << second
56           << ( hour < 12 ? " AM" : " PM" );
57 }
58 // Fig. 6.8: fig06_08.cpp
59 // Demonstrating a default constructor
60 // function for class Time.
61 #include <iostream.h>
62 #include "time2.h"
63
64 int main()
65 {
66     Time t1,           // all arguments defaulted
67         t2(2),         // minute and second defaulted
68         t3(21, 34),    // second defaulted
69         t4(12, 25, 42), // all values specified
70         t5(27, 74, 99); // all bad values specified
71
72     cout << "Constructed with:\n"
73           << "all arguments defaulted:\n  ";
74     t1.printMilitary();
75     cout << "\n  ";
76     t1.printStandard();
77
78     cout << "\nhour specified; minute and second defaulted:"
79           << "\n  ";
80     t2.printMilitary();
81     cout << "\n  ";
82     t2.printStandard();
83

```

```
84     cout << "\nhour and minute specified; second defaulted:"
85         << "\n ";
86     t3.printMilitary();
87     cout << "\n ";
88     t3.printStandard();
89
90     cout << "\nhour, minute, and second specified:"
91         << "\n ";
92     t4.printMilitary();
93     cout << "\n ";
94     t4.printStandard();
95
96     cout << "\nall invalid values specified:"
97         << "\n ";
98     t5.printMilitary();
99     cout << "\n ";
100    t5.printStandard();
101    cout << endl;
102
103    return 0;
104 }
```

**输出结果：**

```
Constructed with:
all arguments defaulted
    00:00
    12:00:00 AM
hour specified; minute and second defaulted:
    02:00
    2:00:00 AM
hour and minute specified; second defaulted:
    21:34
    9:34:00 PM
hour, minute, and second specified:
    12:25
    12:25:42 PM
all invalid values specified:
    00:00
    12:00:00 AM
```

### 6.8 构造函数使用默认参数

在这个程序中，构造函数调用成员函数 `setTime`，将数值传入构造函数（或用默认值）保证 `hour` 值取 0 到 23、`minute` 和 `second` 值取 0 到 59。如果数值超界，则 `setTime` 将其设置为 0（使数据成员保证一致状态）。

注意，`Time` 构造函数也可以写成包含与 `setTime` 成员函数相同的语句。这样可能会使程序更有效，因为不必另外再调用 `setTime` 函数。但让 `Time` 构造函数和 `setTime` 成员函数使用相同代码会使程序维护更加困难。如果 `setTime` 成员函数的实现方法改变，则 `Time` 构造函数的实现方法也要相应改变。`Time` 构造函数直接调用 `setTime` 时，`setTime` 成员函数的实现方法只要改变一次即可，这样就可以在改变实现方法时减少错误。另外，显式声明内联的构造函数或在类定义中定义构造函数也可以提高 `Time` 构造函数的性能（后者隐含内联函数定义）。

**软件工程视点 6.19**

如果类的成员函数已经提供类的构造函数（或其他成员函数）所需功能的所有部分，则从构造函数（或其他成员函数）调用这个成员函数。这样可以简化代码维护和减少修改代码实现方法时的出错机率。因此就形成了一个一般原则：避免重复代码。

**编程技巧 6.6**

只在头文件内的类定义的函数原型中声明默认函数参数值。

**常见编程错误 6.9**

在头文件和成员函数定义中指定同一成员函数的默认初始化值。

图 6.8 的程序初始化五个 Time 对象：一个将三个参数指定为默认值，一个指定一个参数，一个指定两个参数，一个指定三个参数，一个指定三个无效参数。每个对象数据成员均显示实例化和初始化之后的内容。

如果类不定义构造函数，则编译器生成默认构造函数。这种构造函数不进行任何初始化，因此生成对象时，不能保证处于一致状态。

**软件工程视点 6.20**

类不一定有默认构造函数。

## 6.12 使用析构函数

析构函数是类的特殊成员函数。类的析构函数名是类名前面加上代字符（~）。这种命名规则很直观，因为本章稍后将会介绍，代字运算符是按位取反符，从这个意义上，析构函数是构造函数的反函数。

类的析构函数在删除对象时调用，即程序执行离开初始化类对象的范围时。析构函数本身并不实际删除对象，而是进行系统放弃对象内存之前的清理工作，使内存可以复用于保存新对象。

析构函数不接受参数也不返回数值。类只可能有一个析构函数，不能进行析构函数重载。

**常见编程错误 6.10**

向析构函数传递参数、指定析构函数的返回值类型（即使指定 void）、从析构函数返回数值或重载析构函数都是语法错误。

注意，前面介绍的类都没有提供析构函数。下面要介绍几个使用析构函数的例子。第 8 章将介绍析构函数适用于动态分配内存的对象类（例如数组和字符串）。第 7 章将介绍如何动态分配内存和释放内存。

**软件工程视点 6.21**

稍后会介绍，构造函数和析构函数在 C++ 和面向对象编程中相当重要，不是这里的介绍所能说清楚的。

## 6.13 何时调用构造函数与析构函数

构造函数与析构函数是自动调用的。这些函数的调用顺序取决于执行过程进入和离开实例化对象范围的顺序。一般来说，析构函数的调用顺序与构造函数相反。但图 6.9 将介绍对象存储类可以改变析构函数的调用顺序。

全局范围中定义的对象构造函数在文件中的任何其他函数（包括 main）执行之前调用（但不同文件之间全局对象构造函数的执行顺序是不确定的）。当 main 终止或调用 exit 函数时（见第 18 章）调用相应的析构函数。

当程序执行到对象定义时，调用自动局部对象的构造函数。该对象的析构函数在对象离开范围时调用（即离开定义对象的块时）。自动对象的构造函数与析构函数在每次对象进入和离开范围时调用。

static 局部对象的构造函数只在程序执行首次到达对象定义时调用一次，对应的析构函数在 main 终止或调用 exit 函数时调用。

图 6.9 的程序演示了 CreateAndDestroy 类型的对象在几种范围中调用构造函数与析构函数的顺序。程序在全局范围中定义 first，其构造函数在程序开始执行时调用，其析构函数在程序终止时删除所有其他对象之后调用。

```

1 // Fig. 6.9: create.h
2 // Definition of class CreateAndDestroy.
3 // Member functions defined in create.cpp.
4 #ifndef CREATE_H
5 #define CREATE_H
6
7 class CreateAndDestroy {
8 public:
9     CreateAndDestroy( int ); // constructor
10    ~CreateAndDestroy();      // destructor
11 private:
12    int data;
13 };
14
15 #endif
16 // Fig. 6.9: create.cpp
17 // Member function definitions for class CreateAndDestroy
18 #include <iostream.h>
19 #include "create.h"
20
21 CreateAndDestroy::CreateAndDestroy( int value )
22 {
23     data = value;
24     cout << "Object " << data << "   constructor";
25 }
26
27 CreateAndDestroy::~~CreateAndDestroy()
28 { cout << "Object " << data << "   destructor " << endl; }
29 // Fig. 6.9: fig06_09.cpp
30 // Demonstrating the order in which constructors and
31 // destructors are called.
32 #include <iostream.h>
33 #include "create.h"
34
35 void create( void ); // prototype
36
37 CreateAndDestroy first( 1 ); // global object
38
39 int main()
40 {
41     cout << "   (global created before main)" << endl;

```

```

42
43 CreateAndDestroy second( 2 );          // local object
44     cout << "    (local automatic in main)" << endl;
45
46     static CreateAndDestroy third( 3 ); // local object
47     cout << "    (local static in main)" << endl;
48
49     create(); // call function to create objects
50
51     CreateAndDestroy fourth( 4 );        // local object
52     cout << "    (local automatic in main)" << endl;
53     return 0;
54 )
55
56 // Function to create objects
57 void create( void )
58 {
59     CreateAndDestroy fifth( 5 );
60     cout << "    (local automatic in create)" << endl;
61
62     static CreateAndDestroy sixth( 6 );
63     cout << "    (local static in create)" << endl;
64
65     CreateAndDestroy seventh( 7 );
66     cout << "    (local automatic in create)" << endl;
67 }

```

**输出结果:**

```

Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor

```

图 6.9 调用构造函数与析构函数的顺序

函数 main 声明三个对象。对象 second 和 fourth 是局部自动对象，对象 third 是 static 局部对象。这些对象的构造函数在程序执行到对象定义时调用。对象 fourth 和 second 的析构函数在到达 main 结尾时依次调用。由于对象 third 是 static 局部对象，因此到程序结束时才退出，在程序终止时删除所有其他对象之后和调用 first 的析构函数之前调用对象 third 的析构函数。

函数 create 声明三个对象。对象 fifth 和 seventh 是局部自动对象，对象 sixth 是 static 局部对象。对象 seventh 和 fifth 的析构函数在到达 create 结尾时依次调用。由于对象 sixth 是 static 局部对象，因此到程序结束时才退出。sixth 的析构函数在程序终止时删除所有其他对象之后和调用 third 和 first 的析构函数之前调用。

## 6.14 使用数据成员和成员函数

类的 `private` 数据成员只能由类的成员函数（和友元）操作。典型的操作包括用 `computeInterest` 成员函数调整客户的银行借贷（例如 `BankAccount` 类的 `private` 数据成员）。

类通常提供 `public` 成员函数，让类的客户设置（写入）或读取（取得）`private` 数据成员的值。这些函数通常称为 `get` 和 `set`。更具体地说，设置数据成员 `interestRate` 的成员函数通常称为 `setInterestRate`，读取数据成员 `InterestRate` 的值通常称为 `getInterestRate`。读取函数也称为“查询”函数。

提供 `get` 和 `set` 函数与指定数据成员为 `public` 同样重要，这是 C++ 语言在软件工程中的另一优势。如果数据成员为 `public`，则程序中的任何函数可以随意读取和写入这个数据成员。如果数据成员为 `private`，则 `public get` 函数可以让其他函数读取数据，而且数据的显示和格式化也可以用 `get` 函数控制。`public set` 函数通常用于检查数据成员的修改，保证新值是适当的数据项目。例如，如果想把一个月的日期号数设置为 37 会被禁止，将人的身高设置为负值也会被禁止，将数字量设置为字母值也会被拒绝，将一个人的成绩设置为 185 分（取百分制时）同样也会被拒绝等等。

### 软件工程视点 6.22

指定 `private` 数据成员并通过 `public` 成员函数控制这些数据成员的访问（特别是写入访问）可以保证数据的完整性。

### 测试与调试提示 6.5

指定 `private` 数据成员并不能自动保证数据完整性，程序员还要提供验证检查。但 C++ 提供了让程序员方便地设计更好的程序的框架。

### 编程技巧 6.7

设置 `private` 数据值的成员函数应验证所要新值是否正确，如果不正确，则 `set` 函数应将 `private` 数据成员设置为相应的一致状态。

试图要对数据成员指定无效值时，应当提醒类客户。类的 `set` 函数常写成返回一个值，表示试图对数据成员指定无效值。这样就使类的客户可以测试 `set` 函数的返回值，确定其操作的对象是否为有效对象，并在对象无效时采取相应操作。

图 6.10 将 `Time` 类扩展成包括 `private` 数据成员 `hour`、`minute` 和 `second` 的 `get` 和 `set` 函数。`set` 函数严格控制数据成员的设置。如果想把数据成员设置为无效值，则会把数据成员设置为 0（从而使数据成员保持一致状态）。每个 `get` 函数只是返回相应数据成员的值。程序首先用 `set` 函数设置 `Time` 对象 `t` 的 `private` 数据成员为有效值，接着用 `get` 函数读取这个值以便输出。然后 `set` 函数要将 `hour` 和 `second` 成员设置为无效值并将 `minute` 成员设置为有效值，并用 `get` 函数读取这个值以便输出。输出表明，无效值使得数据成员设置为 0。最后，程序将时间设置为 11:58:00 并用函数 `incrementMinutes` 增加 3 分钟。函数 `incrementMinutes` 是个非成员函数，它调用 `get` 和 `set` 成员函数增加 `minute` 成员的值。尽管这样的方法实现了所需的功能，但是多次函数调用降低了程序的性能。下一章将介绍用友元函数消除多次函数调用的性能负担。

### 常见编程错误 6.11

构造函数可以调用类的其他成员函数，如 `set` 和 `get` 函数，但由于构造函数初始化对象，因此数据成员可能还处于不一致状态。数据成员在初始化之前使用可能造成逻辑错误。

```
1 // Fig. 6.10: time3.h
2 // Declaration of the Time class.
3 // Member functions defined in time3.cpp
4
5 // preprocessor directives that
6 // prevent multiple inclusions of header file
7 #ifndef TIME3_H
8 #define TIME3_H
9
10 class Time {
11 public:
12     Time( int = 0, int = 0, int = 0 ); // constructor
13
14     // set functions
15     void setTime( int, int, int ); // set hour, minute, second
16     void setHour( int ); // set hour
17     void setMinute( int ); // set minute
18     void setSecond( int ); // set second
19
20     // get functions
21     int getHour(); // return hour
22     int getMinute(); // return minute
23     int getSecond(); // return second
24
25     void printMilitary(); // output military time
26     void printStandard(); // output standard time
27
28 private:
29     int hour; // 0 - 23
30     int minute; // 0 - 59
31     int second; // 0 - 59
32 };
33
34 #endif
35 // Fig. 6.10: time3.cpp
36 // Member function definitions for Time class.
37 #include "time3.h"
38 #include <iostream.h>
39
40 // Constructor function to initialize private data.
41 // Calls member function setTime to set variables.
42 // Default values are 0 (see class definition).
43 Time::Time( int hr, int min, int sec )
44 { setTime( hr, min, sec ); }
45
46 // Set the values of hour, minute, and second.
47 void Time::setTime( int h, int m, int s )
48 {
49     setHour( h );
50     setMinute( m );
51     setSecond( s );
52 }
53
54 // Set the hour value
55 void Time::setHour( int h )
```



```
56     { hour = ( h >= 0 && h < 24 ) ? h : 0; }
57
58 // Set the minute value
59 void Time::setMinute( int m )
60     { minute = ( m >= 0 && m < 60 ) ? m : 0; }
61
62 // Set the second value
63 void Time::setSecond( int s )
64     { second = ( s >= 0 && s < 60 ) ? s : 0; }
65
66 // Get the hour value
67 int Time::getHour() { return hour; }
68
69 // Get the minute value
70 int Time::getMinute() { return minute; }
71
72 // Get the second value
73 int Time::getSecond() { return second; }
74
75 // Print time in military format
76 void Time::printMilitary()
77 {
78     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
79         << ( minute < 10 ? "0" : "" ) << minute;
80 }
81
82 // Print time in standard format
83 void Time::printStandard()
84 {
85     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
86         << ":" << ( minute < 10 ? "0" : "" ) << minute
87         << ":" << ( second < 10 ? "0" : "" ) << second
88         << ( hour < 12 ? " AM" : " PM" );
89 }
90 // Fig. 6.10: fig06_10.cpp
91 // Demonstrating the Time class set and get functions
92 #include <iostream.h>
93 #include "time3.h"
94
95 void incrementMinutes( Time &, const int );
96
97 int main()
98 {
99     Time t;
100
101     t.setHour( 17 );
102     t.setMinute( 34 );
103     t.setSecond( 25 );
104
105     cout << "Result of setting all valid values:\n"
106         << "   Hour: " << t.getHour()
107         << "   Minute: " << t.getMinute()
108         << "   Second: " << t.getSecond();
109
110     t.setHour( 234 );    // invalid hour set to 0
111     t.setMinute( 43 );
```

```

112     t.setSecond( 6373 ); // invalid second set to 0
113
114     cout << "\n\nResult of attempting to set invalid hour and"
115           << " second:\n  Hour: " << t.getHour()
116           << " Minute: " << t.getMinute()
117           << " Second: " << t.getSecond() << "\n\n";
118
119     t.setTime( 11, 58, 0 );
120     incrementMinutes( t, 3 );
121
122     return 0;
123 }
124
125 void incrementMinutes(Time &tt, const int count)
126 {
127     cout << "Incrementing minute " << count
128           << " times:\nStart time: ";
129     tt.printStandard();
130
131     for ( int i = 0; i < count; i++ ) {
132         tt.setMinute( ( tt.getMinute() + 1 ) % 60);
133
134         if ( tt.getMinute() == 0 )
135             tt.setHour( ( tt.getHour() + 1 ) % 24);
136
137         cout << "\nminute + 1: ";
138         tt.printStandard();
139     }
140
141     cout << endl;
142 }

```

#### 输出结果:

Result of setting all valid values:

Hour: 17 Minute: 34 Second: 25

Result of attempting to set invlid hour and second:

Hour: 0 Minute: 43 Second: 0

Incrementing minute 3 times:

Start time: 11:58:00 AM

minute + 1: 11:59:00 AM

minute + 1: 12:00:00 PM

minute + 1: 12:01:00 PM

图 6.10 使用 set 和 get 函数

从软件工程角度看,使用 set 函数非常重要,因为它们可以进行有效性检查。set 和 get 函数还有其他重要的软件工程优势。

#### 软件工程视点 6.23

通过 set 和 get 函数访问 private 数据不仅能防止数据成员接受无效值,而且还使类的客户不需要考虑数据成员的表达式。这样,如果数据表达式因故改变(通常是为了减少所需存储量或提高性能),只要成员函数提供的接口不变,那么只需改变成员函数而不必改变客户,但客户可能需要重新编译。

## 6.15 微妙的陷阱：返回对 private 数据成员的引用

对象的引用是对象名的别名，因此可以放在赋值语句左边，在这种情况下，引用可以成为可接收赋值的左值。要使用这种功能，就要让类的 public 成员函数返回对该类 private 数据成员的非 const 引用。

图 6.11 用简化的 Time 类演示如何返回 private 数据成员的引用。调用 badSetHour 函数返回的引用作为 private 数据成员 hour 的别名。函数调用可以按任何使用 private 数据成员的方式使用，包括作为赋值语句的左值。

### 编程技巧 6.8

不要让类的 public 成员函数返回对该类 private 数据成员的非 const 引用（或指针），返回这种引用会破坏类的封装。

```

1 // Fig. 6.11: time4.h
2 // Declaration of the Time class.
3 // Member functions defined in time4.cpp
4
5 // preprocessor directives that
6 // prevent multiple inclusions of header file
7 #ifndef TIME4_H
8 #define TIME4_H
9
10 class Time {
11 public
12     Time( int = 0, int = 0, int = 0 );
13     void setTime( int, int, int );
14     int getHour();
15     int &badSetHour( int ); // DANGEROUS reference return
16 private:
17     int hour;
18     int minute;
19     int second;
20 };
21
22 #endif
23 // Fig. 6.11: time4.cpp
24 // Member function definitions for Time class.
25 #include "time4.h"
26 #include <iostream.h>
27
28 // Constructor function to initialize private data.
29 // Calls member function setTime to set variables.
30 // Default values are 0 (see class definition).
31 Time::Time( int hr, int min, int sec )
32 { setTime( hr, min, sec ); }
33
34 // Set the values of hour, minute, and second.
35 void Time::setTime( int h, int m, int s )
36 {
37     hour    = ( h >= 0 && h < 24 ) ? h : 0;
38     minute  = ( m >= 0 && m < 60 ) ? m : 0;
39     second  = ( s >= 0 && s < 60 ) ? s : 0;
40 }

```

```

41
42 // Get the hour value
43 int Time::getHour() { return hour; }
44
45 // POOR PROGRAMMING PRACTICE:
46 // Returning a reference to a private data member.
47 int &Time::badSetHour( int hh )
48 {
49     hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
50
51     return hour; // DANGEROUS reference return
52 }
53 // Fig. 6.11: fig06_11.cpp
54 // Demonstrating a public member function that
55 // returns a reference to a private data member.
56 // Time class has been trimmed for this example.
57 #include <iostream.h>
58 #include "time4.h"
59
60 int main()
61 {
62     Time t;
63     int &hourRef = t.badSetHour( 20 );
64
65     cout << "Hour before modification: " << hourRef;
66     hourRef = 30; // modification with invalid value
67     cout << "\nHour after modification: " << t.getHour();
68
69     // Dangerous: Function call that returns
70     // a reference can be used as an lvalue!
71     t.badSetHour(12) = 74;
72     cout << "\n\n*****\n"
73          << "POOR PROGRAMMING PRACTICE!!!!!!\n"
74          << "badSetHour as an lvalue, Hour: "
75          << t.getHour()
76          << "\n*****" << endl;
77
78     return 0;
79 }

```

#### 输出结果:

```

Hour before modification: 20
Hour after modification: 30

```

```

*****
POOR PROGRAMMING PRACTICE!!!!!!
badSetHour as an lvalue, Hour: 74
*****

```

图 6.11 返回对 private 数据成员引用

程序首先声明 Time 对象 t 和引用 hourRef (把调用 t.badSetHour(20) 返回的引用赋给 hourRef)。程序显示别名 hourRef 的值。然后用这个别名设置 hour 的值为 30 (无效值) 并再次显示该值。最后, 用函数调用本身作为左值并赋值 74 (另一无效值), 再次显示该值。

## 6.16 通过默认的成员复制进行赋值

赋值运算符(=)可以将一个对象赋给另一个相同类型的对象。这种赋值默认通过成员复制(memberwise copy)进行,对象的每个成员一一复制给另一对象的同一成员(如图6.12)(注意:如果类的数据成员包含动态分配存储体,则通过默认的成员复制赋值可能导致严重问题,第8章“运算符重载”中将介绍这些问题及其处理方法)。

对象可以作为函数参数传递并从函数返回。这种传递和返回默认按值调用进行,即传递和返回对象的副本(第8章“运算符重载”中将介绍几个例子)。

### 性能提示 6.4

按值调用传递对象的安全性较好,因为被调函数无法访问原始对象,但如果要复制大对象,则按值调用可能使性能下降。对象按引用调用传递时可以按指针或对象引用传递。按引用调用有性能优势,但安全性较差,因为被调函数可以访问原始对象。按常量引用调用则既安全,又有性能优势。

```
1 // Fig. 6.12: fig06_12.cpp
2 // Demonstrating that class objects can be assigned
3 // to each other using default memberwise copy
4 #include <iostream.h>
5
6 // Simple Date class
7 class Date {
8 public:
9     Date( int = 1, int = 1, int = 1990 ); // default constructor
10     void print();
11 private:
12     int month;
13     int day;
14     int year;
15 };
16
17 // Simple Date constructor with no range checking
18 Date::Date( int m, int d, int y )
19 {
20     month = m;
21     day = d;
22     year = y;
23 }
24
25 // Print the Date in the form mm-dd-yyyy
26 void Date::print()
27 { cout << month << '-' << day << '-' << year; }
28
29 int main()
30 {
31     Date date1( 7, 4, 1993 ), date2; // d2 defaults to 1/1/90
32
33     cout << "date1 = ";
34     date1.print();
35     cout << "\ndate2 = ";
36     date2.print();
37
38     date2 = date1; // assignment by default memberwise copy
```

```
39     cout << "\n\nAfter default memberwise copy, date2 = ";
40     date2.print();
41     cout << endl;
42
43     return 0;
44 }
```

**输出结果:**

```
aatel = 7-4-1993
aate2 = 1-1-1990
```

```
After default memberwise copy, date2 = 7-4-1993
```

图 6.12 通过默认的成员复制将对象赋给另一相同类型的对象

## 6.17 软件复用性

编写面向对象程序的目的是要实现有用的类。类可以通过大量机会获取和分类,让广大程序员使用。许多类库(class library)已经存在,许多类库还在不断开发。人们正在不断推广应用这些类库。软件越来越趋向于从现有的、定义良好、经过认真测试、文档齐全、可移植的各种组件进行构造。这种软件复用性加速了强大的、高质量软件的开发速度。通过复用组件实现快速应用程序开发(rapid applications development, RAD)已经成为一个重要领域。

但还要先解决一些重要问题才能完全实现软件复用性。我们需要有分类机制、许可证机制,用保护机制来保证类的主副本不被搞乱,用描述机制让新系统设计人员能够确定现有对象是否满足其需求,用浏览机制确定有什么类及其与软件开发人员要求的接近程度等等。许多有趣的研究和开发问题需要解决。人们正在积极解决这些问题,因为这种方案的潜在价值是巨大的。

## 6.18 有关对象的思考:编写电梯模拟程序的类

在第1章到第5章“有关对象的思考”中,我们介绍了面向对象的基础,介绍了电梯模拟程序的面向对象设计,第6章介绍了编程和使用C++类的细节。现在就可以开始编写电梯模拟程序的类。

### 电梯实验室任务 4

1. 对第2章到第5章“有关对象的思考”中确定的每个类,编写相应的C++类定义。对每个类,应包括头文件和成员函数定义的源文件。
2. 编写一个驱动程序,测试每个类,并运行完整的电梯模拟程序。注意,可能要等学完第7章之后才能完成电梯模拟程序的工作版本,因此要有耐心,先利用第6章的知识实现电梯模拟程序。第7章将介绍复合,即生成以另一个类为成员的类,这个方法可以表示电梯中的按钮、电铃和门对象为电梯的成员。第7章还要介绍如何用new和delete动态生成和删除对象,帮助生成新人的对象和删除离开的人的对象(在人来和人走时)。
3. 在电梯模拟程序的第一个版本中,只设计简单的面向文本输出,对发生的每个重要事件显示一个消息。程序中的消息可能包括下列字符串:“Person 1 arrives on Floor 1”、“Person 1 presses

Button on Floor 1”、“Elevator arrives on Floor 1”、“Person 1 enters Elevator”等等。注意，建议将消息中表示对象的单词大写。也可以在学完第7章之后再做这个工作。

4. 有的学生还可以用动画图形输出，在屏幕上显示电梯上下移动。

## 小结

- 结构是用其他类型的元素建立的聚合数据类型。
- 结构定义用关键字 `struct` 引入，结构体放在花括号 `{ }` 中，结构定义以分号结尾。
- 结构标志声明结构类型的变量。
- 结构定义并没有在内存中保留任何空间，而是生成新的数据类型，用于声明变量。
- 使用成员访问运算符（包括圆点运算符和箭头运算符）访问结构成员或类成员。圆点运算符通过对象的变量名或对象的引用访问结构和类成员。箭头运算符通过对象指针访问结构和类成员。
- 通过结构生成新数据类型有一定的缺点，可能出现未初始化的数据。如果 `struct` 的实现方法改变，则所有使用这个 `struct` 的程序都要改变。没有保护机制保证数据的正确和保持数据的一致状态。
- 类使程序员可以构造有属性和行为的对象。C++ 中用关键字 `class` 或 `struct` 定义类的类型，通常用关键字 `class`。
- 可以用类名声明该类的对象。
- 类定义以关键字 `class` 开始。类定义体放在左右花括号 `{ }` 之间。类定义用分号终止。
- 任何可以访问类的对象的函数可以访问任何在 `public:` 后面声明的数据成员和成员函数。
- 任何在 `private:` 后面声明的数据成员和成员函数只能由该类的成员函数和友元访问。
- 成员访问说明符总是加上冒号，可以在类定义中按任何顺序多次出现。
- 不能在类的外部访问私有数据。
- 类的实现方法向客户隐藏。
- 构造函数是个特殊成员函数，初始化类对象的数据成员。类的构造函数在生成这个类的对象时自动调用。
- 与类同名而前面加上代字符 `(~)` 的函数称为类的析构函数。
- 类的 `public` 成员函数集称为类的接口或 `public` 接口。
- 在类定义以外定义成员函数时，函数名前面要加上类名和二元作用域运算符 `::`。
- 尽管类定义中声明的成员函数可以在类定义之外定义，但成员函数仍然在类范围内。
- 如果在类定义中定义成员函数，则该成员函数自动成为内联函数，但编译器有权决定其是否作为内联函数。
- 成员函数调用比过程式编程中的传统函数调用更简练，因为成员函数使用的大多数数据可以直接在对象中访问。
- 在类范围内，类成员可以简单地用名字引用。在类范围外，类成员是通过对象名、对象引用和对象指针来引用的。
- 成员选择运算符 `.` 和 `->` 用来访问类成员。
- 良好软件工程的一个基本原则是将接口与实现方法分离。
- 类定义通常放在头文件中，类的成员函数定义放在同一基本名字的源代码文件中。

- 类的默认访问模式是 `private`，因此在类名和第一个说明符（例如 `public:`）之间的所有成员都是 `private` 类型。
- `public` 成员的主要用途是向类的客户提供类的服务。
- 访问类的 `private` 数据应当用称为访问函数的成员函数进行控制。如果类允许客户读取 `private` 数据的值，可以提供一个 `get` 函数；如果类允许客户修改 `private` 数据的值，可以提供一个 `set` 函数。
- 通常将类的数据成员指定为 `private`，将类的成员函数指定为 `public`，可以有助于调试，因为数据操作问题局部化在类成员函数或类的友元中。有些成员函数保持 `private`，是供类中其他函数使用的工具函数。
- 类的数据成员不能在类定义中初始化，应在类的构造函数中初始化或在生成对象之后设置其数值。
- 可以重载构造函数。
- 类对象初始化之后，操作该对象的成员函数应保证对象处于稳定状态。
- 声明类对象时可以提供初始化值，这些初始化值作为参数传递给类的构造函数。
- 构造函数可以指定默认参数。
- 如果类不定义构造函数，则编译器生成默认构造函数。这种构造函数不进行任何初始化，因此生成对象时，不能保证处于一致状态。
- 自动对象的析构函数在对象离开范围时调用，析构函数本身并不删除对象，而是进行系统放弃对象内存之前的清理工作，使内存可以复用于保存新对象。
- 析构函数不接受参数也不返回数值。类只可能有一个析构函数（不能进行析构函数重载）。
- 赋值运算符（`=`）可以将一个对象赋给另一个同类型的对象，这种赋值方式一般通过默认的成员复制来完成。成员复制不适用于所有的类。

## 术语

& reference operator & 引用运算符  
 abstract data type (ADT) 抽象数据类型  
 access function 访问函数  
 arrow member selection operator (`->`) 箭头成员选择运算符  
 attribute 属性  
 behavior 行为  
 binary scope resolution operator (`::`) 二元作用域运算符  
 class  
 class member selector operator (`.`) 类成员选择运算符  
 class definition 类定义  
 class scope 类范围  
 client of a class 类客户  
 consistent state for a data member 数据成员的一致状态

constructor 构造函数  
 data member 数据成员  
 data type 数据类型  
 default constructor 默认构造函数  
 destructor 析构函数  
 dot member selection operator (`.`) 圆点成员选择运算符  
 encapsulation 封装  
 extensibility 可扩展性  
 file scope 文件范围  
 get function get 函数  
 global object 全局对象  
 header file 头文件  
 helper function 帮助函数  
 implementation of a class 类的实现方法  
 information hiding 信息隐藏



|                                         |                                               |
|-----------------------------------------|-----------------------------------------------|
| initialize a class object 初始化类对象        | programmer-defined type 程序员定义类型               |
| inline member function 内联成员函数           | protected                                     |
| instance of a class 类的实例                | proxy class 代理类                               |
| instantiate an object of a class 实例化类对象 | public interface of a class 类的 public 接口      |
| interface to a class 类的接口               | public                                        |
| member access control 成员访问控制            | query function 查询函数                           |
| member access specifiers 成员访问说明符        | rapid applications development (RAD) 快速应用程序开发 |
| member function 成员函数                    | reusable code 可复用代码                           |
| member initializer 成员初始化值               | scope resolution operator (::) 作用域运算符         |
| member selection operator(.和->) 成员选择运算符 | self-referential structure 自引用结构              |
| memberwise copy 成员复制                    | services of a class 类服务                       |
| message 消息                              | set function set 函数                           |
| nonmember function 非成员函数                | software reusability 软件复用性                    |
| nonstatic local object 非静态局部对象          | source-code file 源代码文件                        |
| object 对象                               | static local object 静态局部对象                    |
| object-oriented design (OOD) 面向对象编程     | structure 结构                                  |
| predicate function 判定函数                 | tilde (~) in destructor name 析构函数名中的代字符       |
| principle of least privilege 最低权限原则     | user-defined types 用户自定义类型                    |
| private                                 | utility function 工具函数                         |
| procedural programming 过程式编程            |                                               |

## 自测练习

### 6.1 填空

- 结构定义用关键字 \_\_\_\_\_ 引入。
- 类成员通过 \_\_\_\_\_ 运算符和类对象名或通过 \_\_\_\_\_ 运算符和类对象指针访问。
- 指定为 \_\_\_\_\_ 的类成员只能由类的成员函数和友元访问。
- \_\_\_\_\_ 是个特殊成员函数，用于初始化类的数据成员。
- 类成员的默认访问模式为 \_\_\_\_\_。
- \_\_\_\_\_ 函数用于向类的 private 数据成员赋值。
- \_\_\_\_\_ 可以将一个类对象赋给相同类的另一对象。
- 类的成员函数通常指定为 \_\_\_\_\_，类的数据成员通常指定为 \_\_\_\_\_。
- \_\_\_\_\_ 函数用于读取类的 private 数据值。
- 类的 public 成员函数集称为类的 \_\_\_\_\_。
- 类的实现方法对客户隐藏，也称为 \_\_\_\_\_。
- 关键字 \_\_\_\_\_ 和 \_\_\_\_\_ 可以引入类定义。
- 指定为 \_\_\_\_\_ 的类成员可以在类对象所在范围中的任何位置访问。

### 6.2 寻找下列各题的错误并说明如何纠正。

- 假设 Time 类中声明下列原型：

```
void ~Time( int );
```

b) 下面是 Time 类的部分定义:

```
class Time {
public:
    // function prototypes
private:
    int hour = 0;
    int minute = 0;
    int second = 0;
};
```

c) 假设 Employee 类中声明下列原型:

```
int Employee ( const char *, const char * );
```

## 自测练习答案

- 6.1 a) struct。b) 圆点 (.)、箭头 (->)。c) private。d) 构造函数。e) private。f) set。g) 默认的  
成员复制(用赋值运算符进行)。h) public、private。i) get。j) 接口。k) 封装。l) class、struct。  
m) public。
- 6.2 a) 不正确: 析构函数不能取得参数或返回数值。  
纠正: 删除声明中的返回类型 void 和参数 int。
- b) 不正确: 成员不能在类定义中初始化。  
纠正: 从类定义中删除初始化并在构造函数中初始化数据成员。
- c) 不正确: 构造函数不允许返回数值。  
纠正: 从声明中删除返回类型 int。

## 练习

- 6.3 作用域运算符的作用是什么?
- 6.4 比较 C++ 中的 struct 和 class。
- 6.5 提供一个构造函数, 用 time() 函数中的当前时间初始化 Time 类对象, time() 在 C 标准库  
的头文件 time.h 中声明。
- 6.6 生成 complex 类, 进行复数的运算。编写一个驱动程序, 测试这个类。  
复数的形式如下:

```
realPart + imaginaryPart * i
```

其中 i 为:

$$\sqrt{-1}$$

用浮点变量表示类的 private 数据。提供一个构造函数, 让这个类的对象在声明时初始化。  
不提供初始化值时, 构造函数应包含默认值。对下列情况提供 public 成员函数:

- a) 两个 complex 值相加: 实数部分相加, 虚数部分相加。
- b) 两个 complex 值相减: 实数部分相减, 虚数部分相减。
- c) 打印形如(a, b)的 complex 值, 其中 a 为实数部分, b 为虚数部分。

6.7 生成一个 Rational 类, 进行带分数的运算。编写一个驱动程序, 测试这个类。

用整数变量表示类的 private 数据 (分子和分母), 提供一个对所声明对象初始化的构造函数。不提供初始化值时, 构造函数应包含默认值并将分数存放成简化形式, 例如分数

$$\frac{2}{4}$$

应在对象中存放成分子 1 和分母 2 的形式。对下列情况提供 public 成员函数:

- a) 两个 Rational 值相加, 结果保存成简化形式。
  - b) 两个 Rational 值相减, 结果保存成简化形式。
  - c) 两个 Rational 值相乘, 结果保存成简化形式。
  - d) 两个 Rational 值相除, 结果保存成简化形式。
  - e) 按 a/b 形式打印 Rational 值, 其中 a 为分子, b 为分母。
  - f) 按浮点数形式打印 Rational 值。
- 6.8 修改图 6.10 的 Time 类, 用一个 tick 成员函数将 Time 对象中存放的时间递增 1 秒。Time 对象应该总是保持一致状态。编写一个驱动程序, 在循环中测试 tick 成员函数, 按标准格式打印时间, 从而演示 tick 成员函数的工作情况。要保证测试下列情况:
- a) 递增到下一分钟。
  - b) 递增到下一小时。
  - c) 递增到下一天 (即 11:59:59 PM 到 12:00:00 AM)。
- 6.9 修改图 6.12 的 Date 类, 对数据成员 month、day 和 year 的初始化值进行错误检查。并提供成员函数 nextDay 将日期递增 1 天。Date 对象应该总是保持一致状态。编写一个驱动程序, 在循环中测试 nextDay 成员函数, 按标准格式打印日期, 演示 nextDay 成员函数的工作情况。要保证测试下列情况:
- a) 递增到下一月。
  - b) 递增到下一年。
- 6.10 将练习 6.8 修改的 Time 类和练习 6.9 修改的 Date 类合并成 DateAndTime 类 (第 9 章将介绍继承, 方便地完成这个工作而不必修改现有类定义)。修改 tick 函数, 在时间递增到下一天时调用 nextDay 函数。修改函数 printStandard 和 printMilitary, 不仅输出时间, 也输出日期。编写一个驱动程序, 测试新类 DateAndTime (特别要测试时间递增到下一天时的情况)。
- 6.11 修改图 6.10 的 set 函数, 在 Time 类对象的数据成员要设置为无效值时返回相应错误值。
- 6.12 生成一个 Rectangle 类, 这个类的 length 和 width 属性默认为 1, 其成员函数计算长方形的 perimeter (周长) 和 area (面积)。为该类的 length 和 width 设置 set 和 get 函数。set 函数应验证 length 和 width 均为 0.0 到 20.0 之间的浮点数。
- 6.13 生成比上一练习更复杂的 Rectangle 类, 这个类只保存长方形 4 个角的直角坐标值。构造函数调用一个 set 函数, 接受四组坐标并验证它们均在第一象限中, x、y 坐标均不大于 20.0, 该函数还验证提供的坐标确实构成长方形。成员函数计算 length、width、perimeter 和 area。长度为两个值中较大者。用一个判定函数 square 确定是否为正方形。
- 6.14 修改练习 6.13 的 Rectangle 类, 用 draw 函数在矩形所在第一象限的 25 × 25 闭合框中显示这个长方形。用 setFillCharacter 函数指定绘制长方形外部的字符。用 setPerimeterCharacter 函数指定绘制长方形四边的字符。还可以加上函数对长方形进行比例缩放、旋转和在第一象限指定范围中移动。

- 6.15 生成 `HugeInteger` 类，用 40 个元素的数字数组存放最多 40 位的整数值。提供成员函数 `inputHugeInteger`、`outputHugeInteger`、`addHugeIntegers` 和 `subtractHugeIntegers`。要比较 `HugeInteger` 对象，提供函数 `isEqualTo`、`isNotEqualTo`、`isGreaterThan`、`isLessThan`、`isGreaterThanOrEqualTo` 和 `isLessThanOrEqualTo`，这些判定函数在两个大数间的关系成立时返回 `true`，关系不成立时返回 `false`。该类还提供判定函数 `isZero`，也可以提供成员函数 `multiplyHugeIntegers`、`divideHugeIntegers` 和 `modulusHugeIntegers`。
- 6.16 生成一个 `TicTacToe` 类，编写完成三连棋游戏的程序。这个类包含一个  $3 \times 3$  整型数组作为 `private` 数据，构造函数将空棋盘全部初始化为 0。允许两个人玩游戏。第一个人走棋时，将 1 放在指定格中；第二个人走棋时，将 2 放在指定格中，每次只准移动到空白格。走棋之后，确定是否有人赢了（即 3 个 1 连成一条线或 3 个 2 连成一条线），或是否打了个平手。还可以将程序修改成人机进行游戏，允许游戏者决定他是先走还是后走。甚至可以开发一个程序，在  $4 \times 4 \times 4$  棋盘上玩三维三连棋游戏（注意：这个项目相当难，可能要好几个星期才能完成）。

## 第7章 类与数据抽象（二）

### 教学目标

- 动态生成与删除对象
- 指定 `const` 对象与 `const` 成员函数
- 了解友元函数与友元类的用途
- 了解如何使用 `static` 数据成员和成员函数
- 了解容器类的概念
- 了解遍历容器类元素的迭代类概念
- 了解 `this` 指针的用法

### 7.1 简介

本章继续介绍类与数据抽象。我们要介绍更高级的课题并为第8章介绍类与运算符重载奠定基础。第6章到第8章的讨论鼓励程序员使用对象，我们称之为基于对象编程（object-based programming, OBP）。然后，第9章和第10章介绍继承与多态，这是真正面向对象编程（object-oriented programming, OOP）的技术。本章和后面几章要使用第5章介绍的C语言式字符串，帮助读者掌握C语言指针的复杂课题，以便在工作中处理近二十年来积累的遗留代码。第19章将介绍新的字符串样式，将字符串作为完全成熟的类对象。这样，读者将熟悉C++中生成和操作字符串的两种最主要方法。

### 7.2 `const`（常量）对象与 `const` 成员函数

我们一直强调，最低权限原则（principle of least privilege）是良好软件工程的最基本原则之一。下面介绍这个原则如何应用于对象。

有些对象需要修改，有些不需要。程序员可以用关键字 `const` 指定对象不能修改，且修改时会产生语法错误。例如：

```
const Time noon( 12, 0, 0 );
```

声明 `Time` 类对象 `noon` 为 `const`，并将其初始化为中午 12 时。

#### 软件工程视点 7.1

将对象声明为 `const` 有助于实现最低权限原则，这样试图修改就会产生编译时错误而不是运行时错误。

#### 软件工程视点 7.2

使用 `const` 是正确的类设计、程序设计与编码的关键。

**性能提示 7.1**

声明变量和对象为 `const` 不仅是有效的软件工程做法, 而且能提高性能, 因为如今复杂的优化编译器能对常量进行某些无法对变量进行的优化。

C++ 编译器不允许任何成员函数调用 `const` 对象, 除非该成员函数本身也声明为 `const`, 即使 `get` 成员函数不修改对象时也是这样。声明 `const` 的成员函数不能修改对象, 因为编译器不允许其修改对象。

函数在原型和定义中指定为 `const`, 即在函数参数表和函数定义的左花括号之间插入 `const` 关键字。例如, 下列类 A 的成员函数:

```
int A::getValue() const { return privateDataMember };
```

只是返回一个对象的数据成员值, 可以声明为 `const`。

**常见编程错误 7.1**

定义修改对象数据成员的 `const` 成员函数是个语法错误。

**常见编程错误 7.2**

定义调用同一类实例的非 `const` 成员函数的 `const` 成员函数是个语法错误。

**常见编程错误 7.3**

对 `const` 对象调用非 `const` 成员函数是个语法错误。

**软件工程视点 7.3**

`const` 成员函数可以用非 `const` 版本重载。编译器根据对象是否为 `const` 自动选择所用的重载版本。

这里对构造函数和析构函数产生了一个有趣的问题, 两者都经常需要修改对象。`const` 对象的构造函数和析构函数不需要 `const` 声明。构造函数应允许修改对象, 才能正确地将对象初始化。析构函数应能在对象删除之前进行清理工作。

**常见编程错误 7.4**

将构造函数和析构函数声明为 `const` 是个语法错误。

图 7.1 的程序实例化两个 `Time` 对象, 一个非 `const` 对象和一个 `const` 对象。程序想用非 `const` 成员函数 `setHour` (第 100 行) 和 `printStandard` (第 106 行) 修改 `const` 对象 `noon`。程序还演示了另外三个成员函数调用对象的组合, 一个非 `const` 成员函数调用非 `const` 对象 (第 98 行)、一个 `const` 成员函数调用非 `const` 对象 (第 102 行) 和一个 `const` 成员函数调用 `const` 对象 (第 104 与第 105 行)。输出窗口中显示了一个非 `const` 成员函数调用 `const` 对象时编译器产生的消息。

**编程技巧 7.1**

将所有不需要修改当前对象的成员函数声明为 `const`, 以便在需要时调用 `const` 对象。

```
1 // Fig. 7.1: time5.h
2 // Declaration of the class Time.
3 // Member functions defined in time5.cpp
4 #ifndef TIME5_H
5 #define TIME5_H
6
7 class Time {
```

```
8 public:
9     Time( int = 0, int = 0, int = 0 ); // default constructor
10
11     // set functions
12     void setTime( int, int, int ); // set time
13     void setHour( int ); // set hour
14     void setMinute( int ); // set minute
15     void setSecond( int ); // set second
16
17     // get functions (normally declared const)
18     int getHour() const; // return hour
19     int getMinute() const; // return minute
20     int getSecond() const; // return second
21
22     // print functions (normally declared const)
23     void printMilitary() const; // print military time
24     void printStandard(); // print standard time
25 private:
26     int hour; // 0 - 23
27     int minute; // 0 - 59
28     int second; // 0 - 59
29 };
30
31 #endif
32 // Fig. 7.1: time5.cpp
33 // Member function definitions for Time class.
34 #include <iostream.h>
35 #include "time5.h"
36
37 // Constructor function to initialize private data.
38 // Default values are 0 (see class definition).
39 Time::Time( int hr, int min, int sec )
40 { setTime( hr, min, sec ); }
41
42 // Set the values of hour, minute, and second.
43 void Time::setTime( int h, int m, int s )
44 {
45     setHour( h );
46     setMinute( m );
47     setSecond( s );
48 }
49
50 // Set the hour value
51 void Time::setHour( int h )
52 { hour = ( h >= 0 && h < 24 ) ? h : 0; }
53
54 // Set the minute value
55 void Time::setMinute( int m )
56 { minute = ( m >= 0 && m < 60 ) ? m : 0; }
57
58 // Set the second value
59 void Time::setSecond( int s )
60 { second = ( s >= 0 && s < 60 ) ? s : 0; }
61
62 // Get the hour value
```

```

63 int Time::getHour() const { return hour; }
64
65 // Get the minute value
66 int Time::getMinute() const { return minute; }
67
68 // Get the second value
69 int Time::getSecond() const { return second; }
70
71 // Display military format time: HH:MM:
72 void Time::printMilitary() const
73 {
74     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
75         << ( minute < 10 ? "0" : "" ) << minute;
76 }
77
78 // Display standard format time: HH:MM:SS AM (or PM)
79 void Time::printStandard()
80 {
81     cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":"
82         << ( minute < 10 ? "0" : "" ) << minute << ":"
83         << ( second < 10 ? "0" : "" ) << second
84         << ( hour < 12 ? " AM" : " PM" );
85 }
86 // Fig. 7.1: fig07_01.cpp
87 // Attempting to access a const object with
88 // non-const member functions.
89 #include <iostream.h>
90 #include "time5.h"
91
92 int main()
93 {
94     Time wakeUp( 6, 45, 0 );           // non-constant object
95     const Time noon( 12, 0, 0 );       // constant object
96
97                                     // MEMBER FUNCTION    OBJECT
98     wakeUp.setHour( 18 ); // non-const      non-const
99
100     noon.setHour( 12 ); // non-const        const
101
102     wakeUp.getHour(); // const              non-const
103
104     noon.getMinute(); // const              const
105     noon.printMilitary(); // const          const
106     noon.printStandard(); // non-const      const
107     return 0;
108 }

```

#### 输出结果:

Compiling Fig07\_01.cpp

Fig07\_01.cpp(15) : error: 'setHour' :  
cannot convert 'this' pointer from  
'const class Time' to 'class Time &'  
Conversion loses qualifiers

Fig07\_01.cpp(21) : error: 'printStandard' :  
cannot convert 'this' pointer from



```
'const class Time' to 'class Time &'
Conversion loses qualifiers
```

图 7.1 使用Time类的 const 对象与 const 成员函数

注意, 尽管构造函数应为非 const 成员函数, 但仍然可以对 const 对象调用构造函数。Time 构造函数的定义在第 39 行和第 40 行:

```
Time::Time( int hr, int min, int sec )
{ setTime( hr, min, sec ); }
```

其中Time 构造函数调用另一个非 const 成员函数 setTime 进行Time 对象的初始化。在 const 对象的构造函数调用中调用非 const 成员函数时是合法的。

#### 软件工程视点 7.4

const 对象不能用赋值语句修改, 因此应初始化。类的数据成员声明为 const 时, 要用成员初始化值向构造函数提供类对象数据成员的初始值。

另外注意第 106 行 (源文件中第 21 行):

```
noon.printStandard(); // non-const          const
```

尽管Time 类的成员函数 printStandard 不修改所调用的对象, 但仍然产生一个编译错误。

图 7.2 演示用成员初始化值初始化 Increment 类的 const 数据成员 increment。Increment 的构造函数修改后如下所示:

```
Increment::Increment( int c, int i )
: increment( i )
{ count = c; }
```

符号: increment(i) 将 increment 初始化为数值 i。如果需要多个成员初始化值, 则可以将其放在冒号后面以逗号分隔的列表中。所有数据成员都可以用成员初始化值的语法进行初始化, 但 const 和引用必须用这种方式进行初始化。本章稍后会介绍, 成员对象也要用这种方法进行初始化。第 9 章学习继承时, 会介绍派生类的基类部分也要用这种方法进行初始化。

#### 测试与调试提示 7.1

如果成员函数不修改对象, 则将其声明为 const, 这样可以减少许多错误。

```
1 // Fig. 7.2: fig07_02.cpp
2 // Using a member initializer to initialize a
3 // constant of a built-in data type.
4
5 #include <iostream.h>
6
7 class Increment {
8 public:
9     Increment( int c = 0, int i = 1 );
10    void addIncrement() { count += increment; }
11    void print() const;
12
13 private:
14    int count;
```

```

15     const int increment;          // const data member
16 };
17
18 // Constructor for class Increment
19 Increment::Increment( int c, int i )
20     : increment( i )    // initializer for const member
21 { count = c; }
22
23 // Print the data
24 void Increment::print() const
25 {
26     cout << "count = " << count
27         << ", increment = " << increment << endl;
28 }
29
30 int main()
31 {
32     Increment value( 10, 5 );
33
34     cout << "Before incrementing: ";
35     value.print();
36
37     for ( int j = 0; j < 3; j++ ) {
38         value.addIncrement();
39         cout << "After increment " << j << ": ";
40         value.print();
41     }
42
43     return 0;
44 }

```

**输出结果:**

```

Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5

```

图 7.2 用成员初始化值初始化内部数据类型的常量

图7.3显示的是用赋值语句而不用成员初始化值初始化increment时C++编译器产生的编译错误。

```

1 // Fig. 7.3: fig07_03.cpp
2 // Attempting to initialize a constant of
3 // a built-in data type with an assignment.
4 #include <iostream.h>
5
6 class Increment {
7 public:
8     Increment( int c = 0, int i = 1 );
9     void addIncrement() { count += increment; }
10    void print() const;
11 private:
12    int count;
13    const int increment;
14 };

```

```

15
16 // Constructor for class Increment
17 Increment::Increment( int c, int i )
18 {          // Constant member 'increment' is not initialized
19     count = c;
20     increment = i; // ERROR: Cannot modify a const object
21 }
22
23 // Print the data
24 void Increment::print() const
25 {
26     cout << "count = " << count
27         << ", increment = " << increment << endl;
28 }
29
30 int main()
31 {
32     Increment value( 10, 5 );
33
34     cout << "Before incrementing: ";
35     value.print();
36
37     for ( int j = 0; j < 3; j++ ) {
38         value.addIncrement();
39         cout << "After increment " << j << ": ";
40         value.print();
41     }
42
43     return 0;
44 }

```

**输出结果:**

Compiling...

Fig7\_3.cpp

Fig7\_3.cpp(18) : error: 'increment':  
must be initialized in constructor base/member  
initializer list

Fig7\_3.cpp(20) : error: l-value specifies const object

图 7.3 用赋值语句初始化内部数据类型的常量时产生的编译错误

**常见编程错误 7.5**

不为 const 数据成员提供成员初始化值是个语法错误。

**软件工程视点 7.5**

常量类成员 (const 对象和 const “变量”) 要用成员初始化值的语法初始化, 而不能用赋值语句。

注意第 24 行将 print 函数声明为 const, 但不会有 const 类型的 Increment 对象。

**软件工程视点 7.6**

如果成员函数不修改对象, 最好将其声明为 const。如果不需要生成该类的 const 类型对象, 则这样做是没有必要的。但将这种成员函数声明为 const 有一个好处, 如果不小心修改了这个成员函数中的对象, 则编译器会产生一个语法错误的消息。

### 测试与调试提示 7.2

C++ 之类的语言是不断演变的，新的关键字不断出现。不要用 “object” 之类的标识符。尽管 “object” 目前还不是 C++ 中的关键字，但将来很可能变成关键字，新的编译器可能不能接受现有代码。

C++ 提供了新的关键字 mutable，能够对程序中 const 对象进行处理。第 21 章将介绍关键字 mutable。

## 7.3 复合：把对象作为类成员

AlarmClock 类的对象需要知道何时响铃，因此可以将一个 Time 对象作为类成员，这种功能称为复合（composition）。类可以将其他类对象作为自己的成员。

### 软件工程视点 7.7

复合是软件复用的一种形式，就是一个类将其他类对象作为自己的成员。

生成对象时，自动调用其构造函数，因此要指定参数如何传递给成员对象的构造函数。成员对象按声明的顺序（而不是在构造函数的成员初始化值列表中列出的顺序）并在建立所包含的类对象（也称为宿主对象，host object）之前建立。

图 7.4 用 Employee 类和 Date 类演示一个类作为其他类对象的成员。Employee 类包含 private 数据成员 firstName、lastName、birthDate 和 hireDate。成员 birthDate 和 hireDate 是 Date 类的 const 类型的对象，该 Date 类包含 private 数据成员 month、day 和 year。程序实例化一个 Employee 对象，并初始化和显示其数据成员。注意 Employee 构造函数定义中函数首部的语法：

```
Employee::Employee( char *fname, char *lname,
                    int bmonth, int bday, int byear,
                    int hmonth, int hday, int hyear )
    :birthDate (bmonth, bday, byear),
    hireDate (hmonth, hday, hyear )
```

该构造函数有八个参数（fname、lname、bmonth、bday、byear、hmonth、hday 和 hyear）。首部中的冒号（:）将成员初始化值与参数表分开。成员初始化值指定 Employee 的参数传递给成员对象的构造函数。参数 bmonth、bday 和 byear 传递给 birthDate 构造函数。参数 hmonth、hday 和 hyear 传递给 hireDate 构造函数。多个成员的初始化值用逗号分开。

```
1 // Fig. 7.4: date1.h
2 // Declaration of the Date class.
3 // Member functions defined in date1.cpp
4 #ifndef DATE1_H
5 #define DATE1_H
6
7 class Date {
8 public:
9     Date( int = 1, int = 1, int = 1900 ); // default constructor
10    void print() const; // print date in month/day/year format
11    ~Date(); // provided to confirm destruction order
12 private:
13    int month; // 1-12
14    int day; // 1-31 based on month
```

```
15     int year;    // any year
16
17     // utility function to test proper day for month and year
18     int checkDay( int );
19 };
20
21 #endif
22 // Fig. 7.4: date.cpp
23 // Member function definitions for Date class.
24 #include <iostream.h>
25 #include "date1.h"
26
27 // Constructor: Confirm proper value for month;
28 // call utility function checkDay to confirm proper
29 // value for day.
30 Date::Date( int mn, int dy, int yr )
31 {
32     if ( mn > 0 && mn <= 12 )        // validate the month
33         month = mn;
34     else {
35         month = 1;
36         cout << "Month " << mn << " invalid. Set to month 1.\n";
37     }
38
39     year = yr;                      // should validate yr
40     day = checkDay( dy );           // validate the day
41
42     cout << "Date object constructor for date ";
43     print();                        // interesting: a print with no arguments
44     cout << endl;
45 }
46
47 // Print Date object in form month/day/year
48 void Date::print() const
49 { cout << month << '/' << day << '/' << year; }
50
51 // Destructor: provided to confirm destruction order
52 Date::~Date()
53 {
54     cout << "Date object destructor for date ";
55     print();
56     cout << endl;
57 }
58
59 // Utility function to confirm proper day value
60 // based on month and year.
61 // Is the year 2000 a leap year?
62 int Date::checkDay( int testDay )
63 {
64     static const int daysPerMonth[ 13 ] =
65         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
66
67     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
68         return testDay;
69 }
```

```

70     if ( month == 2 &&          // February: Check for leap year
71         testDay == 29 &&
72         ( year % 400 == 0 ||          // year 2000?
73           ( year % 4 == 0 && year % 100 != 0 ) ) ) // year 2000?
74         return testDay;
75
76     cout << "Day " << testDay << " invalid. Set to day 1.\n";
77
78     return 1; // leave object in consistent state if bad value
79 )
80 // Fig. 7.4: emply1.h
81 // Declaration of the Employee class.
82 // Member functions defined in emply1.cpp
83 #ifndef EMPLOY1_H
84 #define EMPLOY1_H
85
86 #include "date1.h"
87
88 class Employee {
89 public:
90     Employee( char *, char *, int, int, int, int, int, int );
91     void print() const;
92     ~Employee(); // provided to confirm destruction order
93 private:
94     char firstName[ 25 ];
95     char lastName[ 25 ];
96     const Date birthDate;
97     const Date hireDate;
98 };
99
100 #endif
101 // Fig. 7.4: emply1.cpp
102 // Member function definitions for Employee class.
103 #include <iostream.h>
104 #include <string.h>
105 #include "emply1.h"
106 #include "date1.h"
107
108 Employee::Employee( char *fname, char *lname,
109                    int bmonth, int bday, int byear,
110                    int hmonth, int hday, int hyear )
111     : birthDate( bmonth, bday, byear ),
112       hireDate( hmonth, hday, hyear )
113 {
114     // copy fname into firstName and be sure that it fits
115     int length = strlen( fname );
116     length = ( length < 25 ? length : 24 );
117     strncpy( firstName, fname, length );
118     firstName[ length ] = '\0';
119
120     // copy lname into lastName and be sure that it fits
121     length = strlen( lname );
122     length = ( length < 25 ? length : 24 );
123     strncpy( lastName, lname, length );
124     lastName[ length ] = '\0';
125

```

```

126     cout << "Employee object constructor: "
127           << firstName << ' ' << lastName << endl;
128 }
129
130 void Employee::print() const
131 {
132     cout << lastName << ", " << firstName << "\nHired: ";
133     hireDate.print();
134     cout << " Birth date: ";
135     birthDate.print();
136     cout << endl;
137 }
138
139 // Destructor: provided to confirm destruction order
140 Employee::~Employee()
141 {
142     cout << "Employee object destructor: "
143           << lastName << ", " << firstName << endl;
144 }
145 // Fig. 7.4: fig07_04.cpp
146 // Demonstrating composition: an object with member objects.
147 #include <iostream.h>
148 #include "emp1.h"
149
150 int main()
151 {
152     Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );
153
154     cout << '\n';
155     e.print();
156
157     cout << "\nTest Date constructor with invalid values:\n";
158     Date d( 14, 35, 1994 ); // invalid Date values
159     cout << endl;
160     return 0;
161 }

```

**输出结果:**

```

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor Bob Jones

```

```

Jones, Bob
Hired: 3/12/1988 Birth date 7/24/1949
Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994

```

```

Date object destructor for date 1/1/1994
Employee object destructor: Jones, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949

```

图 7.4 使用成员对象的初始化值

记住, `const` 成员和引用也在成员初始化值列表中初始化(第9章会讲到, 派生类的基类部分也是这样初始化的)。Date 类和 Employee 类各有一个析构函数, 分别在删除 Date 和 Employee 对象时打印一个消息。这样就可以从程序输出中确认对象由内向外建立, 由外向内删除(即先删除 Employee 对象, 再删除其中包含的 Date 对象)。

成员对象不需要通过成员初始化值显式初始化。如果不提供成员初始化值, 则隐式调用成员对象的默认构造函数。默认构造函数建立的值(若有)可以用 `set` 函数重定义。

#### 常见编程错误 7.6

没有为成员对象提供初始化值的情况下, 也没有为成员对象提供默认的构造函数, 这样就会产生语法错误。

#### 性能提示 7.2

通过成员初始化值显式初始化成员对象, 这样可以消除两次初始化成员对象的开销, 一次是在调用成员对象的默认构造函数时, 一次是在用 `set` 函数初始化成员对象时。

#### 软件工程视点 7.8

如果一个类用其他类对象作为成员, 则将这个成员对象指定为 `public` 不会破坏该成员对象 `private` 成员的封装与隐藏。

注意第43行调用 Date 成员函数 `print`。C++ 中许多类的成员函数不需要参数。这是因为每个成员函数包含所操作对象的隐式句柄(指针形式)。7.5 节将介绍隐式指针 `this`。

在第一版的 Employee 类中(为了便于编程), 我们用两个 25 字符数组表示 Employee 的姓和名。这些数组如果存储短名称可能浪费内存空间(记住每个数组中有一个字符是字符串的 `null` 终止符 `'\0'`), 超过 24 个字符的姓名要截尾之后才能放得下。本章稍后将介绍另一种形式的 Employee 类, 动态生成适合姓和名的数组长度, 还可以用两个 `string` 对象表示姓名。第19章详细介绍 `string` 标准库类。

## 7.4 友元函数与友元类

类的友元函数(friend function)在类范围之外定义, 但有权访问类的 `private`(和第9章“继承”介绍的受保护)成员。函数或整个类都可以声明为另一个类的友元。

利用友元函数能提高性能, 这里将介绍一个友元函数的例子。本书后面要用友元函数通过类对象和生成迭代类来重载运算符。迭代类对象用于连续选择项目或对容器类(见7.9节)对象中的项目进行操作。容器类对象能够存放项目。成员函数无法进行某些操作时可以使用友元函数(见第8章“运算符重载”)。

要将函数声明为类的友元, 就要在类定义中的函数原型前面加上 `friend` 关键字。要将类 `ClassTwo` 声明为类 `ClassOne` 的友元, 在类 `ClassOne` 的定义中声明如下:

```
friend class classTwo;
```

#### 软件工程视点 7.9

尽管类定义中有友元函数的原型, 但友元仍然不是成员函数。

#### 软件工程视点 7.10

`private`、`protected` 和 `public` 的成员访问符号与友元关系的声明无关, 因此友元关系声明可以放在类定义中的任何地方。



**编程技巧 7.2**

将类中所有友元关系的声明放在类的首部之后, 不要在其前面加上任何成员访问说明符。

友元关系是“给予”的, 而不是“索取”的, 即要让B成为A的友元, A要显式声明B为自己的友元。此外, 友元关系既不对称也不能传递, 例如, 如果A是B的友元, B是C的友元, 并不能说B就是A的友元(不对称)、C就是B的友元或A就是C的友元(不传递)。

**软件工程视点 7.11**

OOP组织中的有些人认为友元关系会破坏信息隐藏和降低面向对象设计方法的价值。

图 7.5 演示了声明与使用友元函数 setX 来设置 Count 类的 private 数据成员 x。注意, 类声明中(习惯上)首先是友元声明, 放在 public 成员函数的声明之前。图 7.6 的程序演示了调用非友元函数 cannotSetX 修改 private 数据成员 x 时编译器产生的消息。图 7.5 和 7.6 介绍了使用友元函数的“结构”, 今后各章会介绍使用友元函数的实际例子。

```
1 // Fig. 7.5: fig07_05.cpp
2 // Friends can access private members of a class.
3 #include <iostream.h>
4
5 // Modified Count class
6 class Count {
7     friend void setX( Count &, int ); // friend declaration
8 public:
9     Count() { x = 0; }                // constructor
10    void print() const { cout << x << endl; } // output
11 private:
12    int x; // data member
13 };
14
15 // Can modify private data of Count because
16 // setX is declared as a friend function of Count
17 void setX( Count &c, int val )
18 {
19     c.x = val; // legal: setX is a friend of Count
20 }
21
22 int main()
23 {
24     Count counter;
25
26     cout << "counter.x after instantiation: ";
27     counter.print();
28     cout << "counter.x after call to setX friend function: ";
29     setX( counter, 8 ); // set x with a friend
30     counter.print();
31     return 0;
32 }
```

**输出结果:**

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

图 7.5 友元可以访问类的 private 成员

注意第 17 行中函数 setX 是 C 语言式的独立函数，而不是 Count 类的成员函数，为此，对 counter 对象调用 setX 时，我们用第 29 行的语句：

```
setX( counter, 8 ); // set x with a friend
```

来提取 counter 参数而不是用句柄（如对象名）调用如下函数：

```
counter.setX( s );
```

#### 软件工程视点 7.12

由于 C++ 是个混合语言，经常在一个程序中并行采用两种函数调用，类 C 语言的调用将基本数据或对象传递给函数，而 C++ 调用将函数（或消息）传递给对象。

```
1 // Fig. 7.6: fig07_06.cpp
2 // Non-friend/non-member functions cannot access
3 // private data of a class.
4 #include <iostream.h>
5
6 // Modified Count class
7 class Count {
8 public:
9     Count() { x = 0; } // constructor
10    void print() const { cout << x << endl; } // output
11 private:
12    int x; // data member
13 };
14
15 // Function tries to modify private data of Count,
16 // but cannot because it is not a friend of Count.
17 void cannotSetX( Count &c, int val )
18 {
19     c.x = val; // ERROR: 'Count::x' is not accessible
20 }
21
22 int main()
23 {
24     Count counter;
25
26     cannotSetX( counter, 3 ); // cannotSetX is not a friend
27     return 0;
28 }
```

#### 输出结果：

Compiling...

Fig07\_06.cpp

Fig07\_06.cpp(19) : error: 'x' :

Cannot access private member declared in class 'Count'

图 7.6 非友元/非成员函数不能访问类的 private 成员

可以指定重载函数为类的友元。每个重载函数如果要作为友元，就要在类定义中显式声明为类的友元。

## 7.5 使用 this 指针

每个对象都可以通过 this 指针访问自己的地址。对象的 this 指针不是对象本身的一部分,即 this 指针不在对该对象进行 sizeof 操作的结果中体现。但 this 指针在每次非 static 成员函数调用对象时 (static 成员见 7.7 节介绍) 作为第一个隐式参数传递给对象 (通过编译器)。

this 指针隐式引用对象的数据成员和成员函数 (当然也可以显式使用)。this 指针的类型取决于对象类型和使用 this 的成员函数是否声明为 const。在 Employee 类的非常量成员函数中, this 指针的类型为 Employee \* const (Employee 对象的常量指针)。在 Employee 类的常量成员函数中, this 指针的类型为 const Employee \* const (为常量 Employee 对象的常量指针)。

下面介绍一个显式使用 this 指针的简单例子,本章稍后和第 8 章将介绍一些使用 this 的复杂例子。每个非 static 成员函数都能访问所调用成员所在对象的 this 指针。

### 性能提示 7.3

为了节约存储空间,每个类的每个成员函数只有一个副本,该类的每个对象都可调用这个成员函数。另一方面每个对象又有自己的类数据成员副本。

图 7.7 演示了显式使用 this 指针,从而使 Test 类的成员函数打印 Test 对象的 private 数据 x。

```
1 // Fig. 7.7: fig07_07.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream.h>
4
5 class Test {
6 public:
7     Test( int = 0 );           // default constructor
8     void print() const;
9 private:
10    int x;
11 };
12
13 Test::Test( int a ) { x = a; } // constructor
14
15 void Test::print() const // () around *this required
16 {
17     cout << "          x = " << x
18         << "\n  this->x = " << this->x
19         << "\n(*this).x = " << ( *this ).x << endl;
20 }
21
22 int main()
23 {
24     Test testObject( 12 );
25
26     testObject.print();
27
28     return 0;
29 }
```

### 输出结果:

```
x = 12
```

```
this->x = 12
(*this).x = 12
```

图 7.7 使用 this 指针

作为演示，图 7.7 中的 print 成员函数首先直接打印 x。然后 print 用两个不同符号通过 this 指针访问 x，一个是 this 指针和箭头运算符 (->)，一个是 this 指针和圆点运算符 (.)。

注意 \*this 和圆点 (成员选择) 运算符一起使用时要用括号括起来。这个括号是必需的，因为圆点运算符的优先级高于 \* 运算符。如果没有括号，则下列表达式：

```
*this.x
```

求值为：

```
*( this.x )
```

这是个语法错误，因为圆点运算符不能和指针一起使用。

#### 常见编程错误 7.7

将对象指针和成员选择运算符 (.) 一起使用是个语法错误，因为成员选择运算符和对象或对该对象的引用一起使用。

this 指针的一个有趣用法是防止对象赋值给自己。第 8 章“运算符重载”中将会介绍，自我赋值可能在对象包含动态分配内存的指针时导致严重的错误。

this 指针的另一用法是允许连续使用成员函数调用。图 7.8 演示了返回 Time 对象的引用，使 Time 类的成员函数调用可以连续使用。成员函数 setTime、setHour、setMinute 和 setSecond 都可以返回 Time & 返回类型的 \*this。

```
1 // Fig. 7.8: time6.h
2 // Cascading member function calls.
3
4 // Declaration of class Time.
5 // Member functions defined in time6.cpp
6 #ifndef TIME6_H
7 #define TIME6_H
8
9 class Time {
10 public:
11     Time( int = 0, int = 0, int = 0 ); // default constructor
12
13     // set functions
14     Time &setTime( int, int, int ); // set hour, minute, second
15     Time &setHour( int ); // set hour
16     Time &setMinute( int ); // set minute
17     Time &setSecond( int ); // set second
18
19     // get functions (normally declared const)
20     int getHour() const; // return hour
21     int getMinute() const; // return minute
22     int getSecond() const; // return second
23
24     // print functions (normally declared const)
25     void printMilitary() const; // print military time
```

```
26 void printStandard() const; // print standard time
27 private:
28     int hour;                // 0 - 23
29     int minute;              // 0 - 59
30     int second;              // 0 - 59
31 };
32
33 #endif
34 // Fig. 7.8: time.cpp
35 // Member function definitions for Time class.
36 #include "time6.h"
37 #include <iostream.h>
38
39 // Constructor function to initialize private data.
40 // Calls member function setTime to set variables.
41 // Default values are 0 (see class definition).
42 Time::Time( int hr, int min, int sec )
43 { setTime( hr, min, sec ); }
44
45 // Set the values of hour, minute, and second.
46 Time &Time::setTime( int h, int m, int s )
47 {
48     setHour( h );
49     setMinute( m );
50     setSecond( s );
51     return *this; // enables cascading
52 }
53
54 // Set the hour value
55 Time &Time::setHour( int h )
56 {
57     hour = ( h >= 0 && h < 24 ) ? h : 0;
58
59     return *this; // enables cascading
60 }
61
62 // Set the minute value
63 Time &Time::setMinute( int m )
64 {
65     minute = ( m >= 0 && m < 60 ) ? m : 0;
66
67     return *this; // enables cascading
68 }
69
70 // Set the second value
71 Time &Time::setSecond( int s )
72 {
73     second = ( s >= 0 && s < 60 ) ? s : 0;
74
75     return *this; // enables cascading
76 }
77
78 // Get the hour value
79 int Time::getHour() const { return hour; }
80
```

```

81 // Get the minute value
82 int Time::getMinute() const { return minute; }
83
84 // Get the second value
85 int Time::getSecond() const { return second; }
86
87 // Display military format time: HH:MM:
88 void Time::printMilitary() const
89 {
90     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
91         << ( minute < 10 ? "0" : "" ) << minute;
92 }
93
94 // Display standard format time: HH:MM:SS AM (or PM)
95 void Time::printStandard() const
96 {
97     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
98         << ":" << ( minute < 10 ? "0" : "" ) << minute
99         << ":" << ( second < 10 ? "0" : "" ) << second
100        << ( hour < 12 ? " AM" : " PM" );
101 }
102 // Fig. 7.8: fig07_08.cpp
103 // Cascading member function calls together
104 // with the this pointer
105 #include <iostream.h>
106 #include "time6.h"
107
108 int main()
109 {
110     Time t;
111
112     t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
113     cout << "Military time: ";
114     t.printMilitary();
115     cout << "\nStandard time: ";
116     t.printStandard();
117
118     cout << "\n\nNew standard time: ";
119     t.setTime( 20, 20, 20 ).printStandard();
120     cout << endl;
121
122     return 0;
123 }

```

**输出结果:**

Military time: 18:30

Standard time: 6:30:22 PM

New standard time: 8:20:20 PM

图 7.8 连续使用成员函数调用

为什么可以将\*`this`作为引用返回呢? 圆点运算符(`.`)的结合律为从左向右, 因此下列表达式:

```
t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
```

首先求值 `t.setHour(18)`，然后返回对象 `t` 的引用，作为这个函数调用的值。其余表达式解释如下：

```
t.setMinute( 30 ).setSecond( 22 );
```

执行 `setMinute(30)`调用并返回 `t` 的等价值，其余表达式解释为：

```
t.setSecond( 22 );
```

注意：

```
t.setTime( 20, 20, 20 ).printStandard();
```

调用也体现了连续使用的特性。这些调用在这个表达式中必须按这个顺序出现，因为类中定义的 `printStandard` 并不返回 `t` 的引用，将上述语句中的 `printStandard` 调用放在 `setTime` 调用之前是个语法错误。

## 7.6 动态内存分配与 `new` 和 `delete` 运算符

`new` 和 `delete` 运算符提供了比 C 语言的 `malloc` 和 `free` 函数调用更好的动态分配内存方法（对任何内部或用户自定义类型）。考虑下列代码：

```
TypeName * typeNamePtr;
```

在 ANSI C 语言中，要为 `TypeName` 类型对象动态分配内存，可以用下列语句：

```
typeNamePtr = malloc( sizeof( TypeName ) );
```

这就要求调用 `malloc` 函数和显式使用 `sizeof` 运算符，在 ANSI C 之前的 C 语言版本中，还要将 `malloc` 返回的指针进行类型转换（`TypeName *`）。`malloc` 函数中没有提供初始化所分配内存块的方法。而在 C++ 中，只要用下列语句：

```
typeNamePtr = new TypeName;
```

`new` 运算符自动生成正确长度的对象并调用对象构造函数和返回正确类型的指针。如果 `new` 无法找到内存空间，则它在 ANSI/ISO C++ 草案标准之前的 C++ 版本中返回 0 指针（注意，第 13 章介绍了如何在 ANSI/ISO C++ 草案标准中处理 `new` 故障。我们要显示 `new` 如何“抛出”异常，如何“捕获”与处理异常）。要在 C++ 中释放这个对象的空间，就要用 `delete` 运算符，如下所示：

```
delete typeNamePtr;
```

C++ 允许对新生成的对象提供初始化值，如下所示：

```
float *thingPtr = new float ( 3.14159 );
```

将新生成的对象 `float` 初始化为 3.14159。

可以生成 10 个元素的整型数组并赋给 `arrayPtr`，如下所示：

```
int *arrayPtr = new int[ 10 ];
```

这个数组可以用下列语句删除：

```
delete [] arrayPtr;
```

可以看出,使用new和delete而不用malloc和free还有其他好处。new自动调用构造函数,delete自动调用析构函数。

#### 常见编程错误 7.8

将new和delete动态分配内存的方法与malloc和free动态分配内存的方法混合使用是个逻辑错误:malloc分配的空间无法用delete释放,new生成的对象无法用free删除。

#### 常见编程错误 7.9

用delete而不是delete[]删除数组可能导致运行时的逻辑错误。为了避免这个问题,数组生成的内存空间要用delete[]运算符删除,各个元素生成的内存空间要用delete运算符删除。

#### 编程技巧 7.3

C++程序也可以包含用malloc生成和用free删除的存储空间以及用new生成和用delete删除的对象。但最好还是使用new和delete。

## 7.7 static 类成员

类的每个对象有自己的所有数据成员的副本,有时类的所有对象应共享变量的一个副本,因此可以使用static类变量。static类变量表示的是类范围中所有对象共享的信息。static类成员的声明以static关键字开始。

下面用一个视频游戏的例子说明static类共享数据的作用。假设视频游戏中有Marian和其他太空人。每个Marian都很勇敢,只要有5个Marian存在,就可以攻击其他太空人。如果Marian的人数不到5个,则不能进行攻击。因此每个Marian都要知道marianCount。我们在Marian类中提供一个marianCount数据成员,这样,每个Marian有该数据成员的副本,每次生成新Marian时,都要更新每个Marian中的marianCount,这样既浪费空间又浪费时间。为此,我们将marianCount声明为static,这样就使marianCount成为类中共享的数据。每个Marian都可以访问marianCount,就像是自己的数据成员一样,但C++只需维护marianCount的一个静态副本,这样可以节省空间。让Marian构造函数递增静态marianCount还能节省时间,因为只有一个副本,不需要对每个Marian对象递增marianCount。

#### 性能提示 7.4

如果一个数据副本就足够使用,用static数据成员可以节省存储空间。

尽管static数据成员像是全局变量,但static数据成员的作用域是类范围,static成员可以为public、private或protected。static数据成员只在文件范围中初始化一次。类的public static类成员可以通过类的任何对象访问,也可以用二元作用域运算符通过类名访问。类的private和protected static成员应通过类的public成员函数或通过类的友元访问。即使类没有一个对象,其static成员依然存在。要在类没有一个对象时访问public static类成员,只需在成员数据名前加上类名和二元作用域运算符(::)。要在类没有一个对象时访问protected static类成员,则需提供一个public static成员函数,并在调用函数时在函数名前面加上类名和二元作用域运算符。

图7.9的程序演示了private static数据成员和public static成员函数的使用。数据成员count在文件范围内初始化为0,如下所示:

```
int Employee::count = 0
```



数据成员 `count` 维护 `Employee` 类实例化的对象个数。`Employee` 类的对象存在时, 可以通过 `Employee` 对象的任何成员函数引用成员 `count`, 本例中, 构造函数和析构函数都引用了 `count`。

#### 常见编程错误 7.10

在文件范围的 `static` 类变量定义中包括 `static` 关键字是个语法错误。

```

1 // Fig. 7.9: employ1.h
2 // An employee class
3 #ifndef EMPLOY1_H
4 #define EMPLOY1_H
5
6 class Employee {
7 public:
8     Employee( const char*, const char* ); // constructor
9     ~Employee(); // destructor
10    const char *getFirstName() const; // return first name
11    const char *getLastName() const; // return last name
12
13    // static member function
14    static int getCount(); // return # objects instantiated
15
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data member
21    static int count; // number of objects instantiated
22 };
23
24 #endif
25 // Fig. 7.9: employ1.cpp
26 // Member function definitions for class Employee
27 #include <iostream.h>
28 #include <string.h>
29 #include <assert.h>
30 #include "employ1.h"
31
32 // Initialize the static data member
33 int Employee::count = 0;
34
35 // Define the static member function that
36 // returns the number of employee objects instantiated.
37 int Employee::getCount() { return count; }
38
39 // Constructor dynamically allocates space for the
40 // first and last name and uses strcpy to copy
41 // the first and last names into the object
42 Employee::Employee( const char *first, const char *last )
43 {
44     firstName = new char[ strlen( first ) + 1 ];
45     assert( firstName != 0 ); // ensure memory allocated
46     strcpy( firstName, first );
47
48     lastName = new char[ strlen( last ) + 1 ];

```

---

```

49  assert( lastName != 0 );    // ensure memory allocated
50  strcpy( lastName, last );
51
52  ++count; // increment static count of employees
53  cout << "Employee constructor for " << firstName
54       << ' ' << lastName << " called." << endl;
55 }
56
57 // Destructor deallocates dynamically allocated memory
58 Employee::~Employee()
59 {
60     cout << "~Employee() called for " << firstName
61          << ' ' << lastName << endl;
62     delete [] firstName; // recapture memory
63     delete [] lastName;  // recapture memory
64     --count; // decrement static count of employees
65 }
66
67 // Return first name of employee
68 const char *Employee::getFirstName() const
69 {
70     // Const before return type prevents client modifying
71     // private data. Client should copy returned string before
72     // destructor deletes storage to prevent undefined pointer.
73     return firstName;
74 }
75
76 // Return last name of employee
77 const char *Employee::getLastName() const
78 {
79     // Const before return type prevents client modifying
80     // private data. Client should copy returned string before
81     // destructor deletes storage to prevent undefined pointer.
82     return lastName;
83 }
84 // Fig. 7.9: fig07_09.cpp
85 // Driver to test the employee class
86 #include <iostream.h>
87 #include "employ1.h"
88
89 int main()
90 {
91     cout << "Number of employees before instantiation is "
92          << Employee::getCount() << endl; // use class name
93
94     Employee *e1Ptr = new Employee( "Susan", "Baker" );
95     Employee *e2Ptr = new Employee( "Robert", "Jones" );
96
97     cout << "Number of employees after instantiation is "
98          << e1Ptr->getCount();
99
100    cout << "\n\nEmployee 1: "
101          << e1Ptr->getFirstName()
102          << " " << e1Ptr->getLastName()
103          << "\nEmployee 2: "

```

```

104         << e2Ptr->getFirstName()
105         << " " << e2Ptr->getLastName() << "\n\n";
106
107     delete e1Ptr;    // recapture memory
108     e1Ptr = 0;
109     delete e2Ptr;    // recapture memory
110     e2Ptr = 0;
111
112     cout << "Number of employees after deletion is "
113           << Employee::getCount() << endl;
114
115     return 0;
116 }

```

**输出结果:**

```

Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

```

```

Employee 1: Susan Baker
Employee 2: Robert Jones

```

```

~ Employee() called for Susan Baker
~ Employee() called for Robert Jones
Number of employees after deletion is 0

```

图 7.9 用 static 数据成员维护类的对象个数

Employee类的对象不存在时,仍然可以引用成员 count,但只能通过调用static成员函数getCount:

```
Employee::getCount()
```

本例中,函数getCount确定当前实例化的Employee对象个数。注意,程序中没有实例化的对象时,发出Employee::getCount()函数调用。但如果有实例化的对象,则可以通过一个对象调用函数getCount,见第97行和第98行的语句:

```

cout << "Number of employees after instantiation is "
      << e1Ptr->getCount();

```

注意,调用e2Ptr->getCount()和Employee::getCount()也能使上述语句运行。

**软件工程视点 7.13**

有些公司的软件工程标准要求所有static成员函数只能对类名句柄调用,而不能对对象句柄调用。

如果成员函数不访问非static类数据成员和成员函数,则可以声明为static。与非static成员函数不同的是,static成员函数没有this指针,因为static类数据成员和成员函数是独立于类对象而存在的。

**常见编程错误 7.11**

在static成员函数中引用this指针是个语法错误。

#### 常见编程错误 7.12

将 static 成员函数声明为 const 是个语法错误。

#### 软件工程视点 7.14

即使在类没有实例化任何对象时，类的 static 数据成员和成员函数就已经存在并可使用。

第 94 行和第 95 行用运算符 new 动态分配两个 Employee 对象。分配每个 Employee 对象时调用其构造函数。第 107 行和第 109 行用 delete 释放两个 Employee 对象的内存空间时，调用其析构函数。

#### 编程技巧 7.4

删除动态分配内存后，设置指向该内存的指针指向 0，这样就切断了指针与前面所分配内存的连接。

注意 Employee 构造函数中使用了 assert。assert 宏在 assert.h 头文件中定义，测试条件值。如果表达式值为 false，则 assert 发出错误消息，并调用 abort 函数（在一般实用程序头文件 stdlib.h 中）终止程序执行。这是个有用的调试工具，可以测试变量是否有正确值。在这个程序中，assert 确定 new 运算符能否满足动态分配内存的请求。例如，在 Employee 构造函数中，下列语句（也称为断言）：

```
assert( firstName !=0 );
```

测试指针 firstName 以确定其是否不等于 0。如果上述语句中的条件为 true，则程序继续执行，不被中断。如果上述语句中的条件为 false，则程序打印一个错误消息，包括行号、测试条件和断言所在的文件名，然后程序终止。程序员可以从这个代码区域找出错误。第 13 章“异常处理”中将介绍处理执行时错误的更好方法。

断言不一定要在调试完成后删除。程序不再用这个断言进行调试时，只要在程序文件开头插入下列语句即可：

```
#define NDEBUG
```

这时预处理程序忽略所有断言而不必由程序员手工删除每条断言。

注意函数 getFirstName 和 getLastName 的实现方法向类的客户返回常量字符指针。在这个实现方法中，如果客户要保留姓和名的副本，则客户要在取得对象的常量字符指针之后负责复制 Employee 对象的动态分配内存。注意，还可以使用 getFirstName 和 getLastName 让客户向每个函数传递字符数组和数组长度。然后函数可以将姓或名复制到客户提供的字符数组中。

## 7.8 数据抽象与信息隐藏

类通常对类的客户隐藏其实现细节，即所谓的信息隐藏。下列以堆栈数据结构作为信息隐藏的例子。

可以把堆栈看成一堆盘子。将盘子放在堆中时，总是放在顶部（压入堆栈）。从堆中取下盘子时，总是从顶上取（称为弹出堆栈）。堆栈是后进先出（last-in, first-out; LIFO）的数据结构，最后放进堆栈的项目最先从堆栈中取出。

程序员可以生成堆栈类，对客户隐藏实现细节。堆栈可以方便地用数组实现（或用第 15 章“数据结构”中的链表）。堆栈类的客户不需要知道堆栈如何实现，只要求数据项目按后进先出的方式处理即可。描述类的功能而不管其实现细节称为数据抽象（data abstraction），C++ 的类定义了抽象数据类型（abstract data types, ADT）。尽管用户可能知道类的实现细节，但编码时不能依赖于这些

实现细节。只要该类的 public 接口不变, 类的实现细节(如堆栈的实现和“压入”“弹出”的操作)可能改变而不影响系统其余部分。

高级语言的任务是生成便于程序员使用的视图。由于没有一个标准视图, 因此编程语言种类很多。C++ 中的面向对象编程显示了另一种视图。

大多数编程语言都强调操作。在这些语言中, 数据的存在只是为了支持程序所需的操作。数据比操作更不重要, 数据是原始的, 只有几个内部数据类型, 程序员很难生成自己的新数据类型。

C++ 和面向对象编程中改变了这些观点。C++ 提高了数据的重要性。C++ 中的主要活动就是生成自己的新数据类型(即类)和表达这些数据类型之间的相互作用。

要转向这个方向, 编程语言组织要规范数据的一些概念。我们考虑的规范化就是抽象数据类型。抽象数据类型就像十多年前的结构化编程一样备受关注, 抽象数据类型并不能代替结构化编程, 只是提供了其他规范, 可以进一步改善程序开发过程。

什么是抽象数据类型呢? 考虑内部类型 int, 我们看到它是数学中的整数, 但 int 在计算机中并不完全是数学中的整数, 计算机中的 int 长度是很有限的。例如, 32 位机器上的 int 只限于 -20 亿到 20 亿之间。如果计算结果超出这个范围, 则会发生溢出错误, 机器会以机器相关的方式响应, 可能悄悄地产生错误结果, 而数学中的整数则没有这个问题。因此, 计算机中的 int 概念实际上只是实际整数的一个近似, float 也是这样。

char 同样是个近似, char 值通常是 8 位模式的 0 和 1, 这些模式与所表示的字符(如 Z、小写 z、美元号 \$、数字 5 等等)完全不同。char 类型的值在大多数计算机上都是很有限的。7 位 ASCII 字符集只提供 128 个不同字符值。这显然不足以表达中文、日文等需要成千上万个字符的语言。

由此可见, 即使 C++ 之类的编程语言提供的内部数据类型, 实际上也只是实际生活中概念和行为的近似或模型。前面我们一直理所当然地使用 int, 现在则有了全新的概念。int、float、char 之类的类型都是抽象数据类型, 实际上是在计算机系统中一定程度地近似表示实际中的概念。

抽象数据类型实际上包含两个概念, 即数据表达(data representation)和该数据允许的操作(operation)。例如, int 的概念定义了 C++ 中的加、减、乘、除、求模操作, 但除数为 0 时则未定义, 这种操作与机器参数有关, 如计算机系统的定长字长度。另一个例子是负整数的概念, 其运算和数据表达是可应用的, 但负整数的平方根则没有定义。C++ 中程序员用类实现抽象数据类型。我们将在第 12 章“模板”中生成自己的堆栈类, 并在第 20 章“标准模板库(STL)”中介绍标准库 stack 类。

### 7.8.1 范例: 数组抽象数据类型

第 4 章曾介绍过数组。数组就是一个指针和一些内存空间。如果程序员小心谨慎, 则利用这些原始功能就可以进行数组操作。数组还有许多精彩的操作, 但 C++ 中没有提供。利用 C++ 类, 程序员可以开发比“原始”数组更精彩的数组 ADT。数组类可以提供许多有用的功能, 例如:

- 下标范围检查。
- 任意范围下标而不一定从 0 开始。
- 数组赋值。
- 数组比较。
- 数组输入/输出。
- 已知长度数组。

我们将在第 8 章“运算符重载”中生成自己的数组类, 并在第 20 章“标准模板库(STL)”中介绍标准库 vector 类。

C++ 有少量内部类型，类可以扩展这个基础编程语言。

#### 软件工程视点 7.15

程序员可以通过类机制生成新类型。这些新类型可以方便地像内部类型一样使用。这样，C++ 是个可扩展的语言。尽管这个语言可以方便地用新类型扩展，但基础语言本身不能改变。

C++ 环境中生成的新类可以专属一个人、一个小组或一个公司。类也可以放进标准类库中，以便推广使用。这不一定能上升为标准，但事实上标准正在出现。C++ 的全部价值只有在充实而标准化的类库得到广泛应用时才能完全体现。在美国，ANSI（美国国家标准协会）正在进行这种标准化。ANSI 和 ISO（国际标准化组织）正在开发 C++ 的标准版本。学习 C++ 和面向对象编程的读者可以利用丰富的库实现快速的面向组件的软件开发。

### 7.8.2 范例：字符串抽象数据类型

C++ 是一种定义简练的语言，只向程序员提供了建立各种系统的原始功能。这个语言保证了最小编程负担。C++ 适合应用编程和系统编程，后者对程序的性能有更高要求。当然，C++ 内部数据类型中也可以包括字符串数据类型，但这个语言设计成包括通过类生成和实现字符串抽象数据类型的机制。我们将在第 8 章开发自己的字符串 ADT。ANSI/ISO 草案标准中有一个 `string` 类，将在第 19 章详细介绍。

### 7.8.3 范例：队列抽象数据类型

我们经常遇到排队，等待的队称为队列（queue）。我们排队在超市结账，排队换煤气，排队上公共汽车，排队在高速公路上交费，学生注册时和到食堂买饭时都要排队。计算机系统内部也使用许多排队，因此，我们需要编写一个模拟排队的程序。

队列是个抽象数据类型的范例。队列向客户提供了明确的行为。客户一次一件地将东西放进队列中，称为进队（enqueue），然后从队列中一次一件地取出东西，称为出队（dequeue）。理论上，队列可以无限长，但真正的队列是有限的。队列中的项目按先进先出（first-in, first-out; FIFO）顺序出列，第一个插入队列的项目第一个离开队列。

队列隐藏了内部数据表示，跟踪队列中的当前项目，为客户提供一组操作，如入队和出队。客户并不关心队列的实现，只要求队列正常操作。客户让一个新项目入队时，队列应接受这个项目，并放在某种先进先出的数据结构中。客户要从队列中取一个项目时，应从内部表示中删除这个项目，并将该项目按先进先出的顺序传递给外界（即队列的客户），下一次出队的项目应是队列中等待时间最长的项目。

队列 ADT 保证内部数据结构的完整性。客户不能直接操作这个数据结构，只有队列 ADT 能访问其内部数据。客户只能对数据表达进行允许的操作，ADT 的 `public` 接口中不提供的操作将由 ADT 以适当方式拒绝，例如发一个错误消息、终止执行或忽略操作请求。

第 15 章“数据结构”中将建立自己的队列类，第 20 章将介绍标准库 `queue` 类。

## 7.9 容器类与迭代

最常见的类型包括容器类（container class），也称集合类（collection class），是保存一组对象集合的类。容器类通常提供插入、删除、查找、排序和测试类成员项目等操作。数组、堆栈、队列、树和链表都是容器类，第 4 章介绍了数组，第 15 章和 20 章将介绍其他数据结构。

容器类经常与迭代对象 (iterator object; 或简称迭代器, iterator) 相关联。迭代对象返回集合中的下一个项目 (或对集合中的下一个项目进行某种操作)。编写类的迭代器之后, 要取得类中的下一个元素很简单, 迭代器通常指定为类的友元, 以提高性能, 使迭代器能通过迭代直接访问 `private` 数据。就像几个人共读的书中可以插好几个标签一样, 可以有同时操作几个迭代器的容器类, 每个迭代器包含自己的位置信息。第 20 章 “标准模板库 (STL)” 中将详细介绍容器和迭代器。

## 7.10 代理类

通过隐藏类的实现细节可以防止访问类中的专属信息 (包括 `private` 数据) 和专属程序逻辑。向客户提供代理类 (proxy class), 代理类只能访问类的 `public` 接口, 这样就可以让客户使用类的服务而不必让客户访问类的实现细节。

实现代理类需要几个步骤 (如图 7.10)。首先, 我们生成要隐藏 `private` 数据的类的类定义和实现文件。我们的例子使用 `Implementation` 类、代理类 `Interface` 和测试程序, 并输出了结果。

`Implementation` 类提供一个 `private` 数据成员 `value` (这是要对客户隐藏的数据)、一个初始化 `value` 的构造函数以及函数 `setValue` 和 `getValue`。

```

1 // Fig. 7.10: implementation.h
2 // Header file for class Implementation
3
4 class Implementation {
5     public:
6         Implementation( int v ) { value = v; }
7         void setValue( int v ) { value = v; }
8         int getValue() const { return value; }
9
10    private:
11        int value;
12 };
13 // Fig. 7.10: interface.h
14 // Header file for interface.cpp
15 class Implementation; // forward class declaration
16
17 class Interface {
18     public:
19         Interface( int );
20         void setValue( int ); // same public interface as
21         int getValue() const; // class Implementation
22     private:
23         Implementation *ptr; // requires previous
24                               // forward declaration
25 };
26 // Fig. 7.10: interface.cpp
27 // Definition of class Interface
28 #include "interface.h"
29 #include "implementation.h"
30
31 Interface::Interface( int v )
32     : ptr ( new Implementation( v ) ) { }
33

```

```

34 // call Implementation's setValue function
35 void Interface::setValue( int v ) { ptr->setValue( v ); }
36
37 // call Implementation's getValue function
38 int Interface::getValue() const { return ptr->getValue(); }
39 // Fig. 7.10: fig07_10.cpp
40 // Hiding a class's private data with a proxy class.
41 #include <iostream.h>
42 #include "interface.h"
43
44 int main()
45 {
46     Interface i( 5 );
47
48     cout << "Interface contains: " << i.getValue()
49         << " before setValue" << endl;
50     i.setValue( 10 );
51     cout << "Interface contains: " << i.getValue()
52         << " after setValue" << endl;
53     return 0;
54 }

```

**输出结果：**

```

Interface contains: 5 before setVal
Interface contains: 10 after setVal

```

图 7.10 实现代理类

我们用 Implementation 类的同一个 public 接口生成代理类的定义。代理类的惟一 private 成员是 Implementation 类对象的指针。利用指针可以隐藏 Implementation 类的实现细节。

图 7.10 的 Interface 类是 Implementation 类的代理类。注意 Interface 类中提到 Implementation 类时只有第 23 行的指针声明。类定义（如 Interface 类）只使用另一个类（如 Implementation 类）的指针时，另一个类的头文件（通常显示该类的 private 数据）不需要用 #include 包含在内。只要在文件中使用该类型之前用提前类声明（forward class declaration）将另外的这个类声明为一种数据类型即可（见第 15 行）。

实现文件包含代理类 Interface 的成员函数，这是惟一包含 Implementation 类所在头文件 implementation.h 的文件。文件 interface.cpp 以预编译对象文件形式和头文件 interface.h 一起提供给客户，该头文件包含代理类提供服务的函数原型。由于文件 interface.cpp 只以已编译对象文件形式提供给客户，因此客户无法看到代理类与专属类之间的交互。

图 7.10 的程序测试 Interface 类。注意，main 中只包含 Interface 类的头文件，而没有提到 Implementation 类。因此，客户根本不知道 Implementation 类的 private 数据。

## 7.11 有关对象的思考：在电梯模拟程序中使用复合和动态对象管理

第 2 章到第 5 章设计了电梯模拟程序，第 6 章开始了电梯模拟程序的编程。第 7 章中介绍了实现可投入使用的完整电梯模拟程序所需的其他技术，包括动态对象管理技术，用 new 和 delete 生成和删除模拟程序执行时所需的对象。我们还介绍了复合，从而可以在一个类中包含其他类对象成员。通过复合可以建立大楼类，包含电梯和层，并可建立电梯类，包含按钮、门和电铃。



### 电梯实验室任务5

1. 每当有另一个人进入时,用 new 生成新的 Person 对象,表示这个人。注意 new 调用所生成对象的构造函数,这个构造函数应初始化该对象。每次有人离开时,用 delete 删除 Person 对象并释放该对象占用的存储空间,delete 调用所删除对象的析构函数。
2. 枚举电梯模拟程序中所实现的类之间的复合关系。修改第6章“有关对象的思考”一节生成的类定义,反映这种复合关系。
3. 完成模拟程序的实现工作。后面各章会建议如何改进模拟程序。

### 小结

- 关键字 const 指定的对象不能修改。
- C++ 编译器不允许任何非 const 成员函数调用 const 对象。
- 试图通过类的 const 成员函数修改该类对象的数据成员是个语法错误。
- 函数在原型和定义中指定为 const。
- const 成员函数可以用非 const 版本重载。编译器根据对象是否声明为 const 自动选择所用的重载版本。
- const 对象应初始化。要用成员初始化值向构造函数提供类对象数据成员的初始值。
- 类可以将其他类对象作为类成员。
- 成员对象按声明的顺序在构造所在类对象之前构造。
- 如果不提供成员初始化值,则隐含调用成员对象的默认构造函数。
- 类的友元函数在类范围之外定义,但有权访问类的所有成员。
- 友元关系声明可以放在类定义中的任何地方。
- this 指针隐式引用对象的数据成员和成员函数。
- 每个对象都可以通过 this 指针访问自己的地址。
- this 指针也可以显式使用。
- new 运算符自动生成正确长度的对象,调用对象构造函数和返回正确类型的指针。要在 C++ 中释放这个对象的空间,需要使用 delete 运算符。
- 对象数组可以用 new 动态分配,如下所示:

```
int *ptr = new int[100];
```

分配 100 个整数的数组并将数组开始位置指定为 ptr。上述整数数组可以用下列语句删除:

```
delete [] ptr;
```

- static 类变量表示整个类范围的信息。static 类成员的声明以 static 关键字开始。
- static 数据成员的作用域是类范围。
- 类的 public static 类成员可以通过类的任何对象访问,也可以用二元作用域运算符通过类名访问。
- 如果成员函数不访问非 static 类数据成员和成员函数,则可以声明为 static。与非 static 成员函数不同的是,static 成员函数没有 this 指针,因为 static 类数据成员和成员函数是独立于类对象而存在的。
- 类通常对类的客户隐藏实现细节,这称为信息隐藏。
- 堆栈是后进先出(LIFO)的数据结构,最后放进堆栈的项目最先从堆栈中取出。

- 描述类的功能而不管其实现细节称为数据抽象, C++ 类定义所谓的抽象数据类型 (ADT)。
- C++ 提高了数据的重要性。C++ 中的主要活动就是生成自己的新数据类型 (即类) 和表达这些数据类型之间的相互作用。
- 抽象数据类型实际上是在计算机系统中一定程度地近似表示现实世界的概念。
- 抽象数据类型实际上包含两个概念, 即数据表达和该数据允许的操作。
- C++ 是可扩展的语言。尽管这个语言可以方便地用新类型扩展, 但基础语言本身不能改变。
- C++ 是一种定义简练的语言, 只向程序员提供了建立各种系统的原始功能。这种语言保证了最小编程负担。
- 队列中的项目按先进先出 (FIFO) 顺序出队, 第一个插入队列的项目第一个离开队列。
- 容器类是保存一组对象集合的类。容器类通常提供插入、删除、查找、排序和测试类成员项目等操作。
- 容器类经常与迭代对象 (或简称迭代器) 相关联。迭代对象返回集合中的下一个项目 (或对集合中的下一个项目进行某种操作)。
- 向客户提供代理类, 代理类只能访问类的 public 接口, 这样就可以让客户使用类的服务而不必让客户访问类的实现细节。
- 代理类惟一的 private 成员是隐藏该类对象的 private 数据的指针。
- 类定义只使用另一个类的指针时, 另一个类的头文件 (通常显示该类的 private 数据) 不需要用 #include 包含在内。只要在文件中使用该类型之前用提前类声明将另外的这个类声明为一种数据类型即可。
- 实现文件包含代理类成员函数, 该文件包含隐藏该类 private 数据的头文件。
- 实现文件以预编译对象文件形式和头文件一起提供给客户, 该头文件包含代理类提供服务的函数原型。

## 术语

abstract data type (ADT) 抽象数据类型  
 binary scope resolution operator (::  
 运算符  
 cascading member function calls 连续使用成员  
 函数调用  
 class scope 类范围  
 composition 复合  
 const member function const 成员函数  
 const object const 对象  
 constructor 构造函数  
 container 容器  
 data representations 数据表达  
 default constructor 默认构造函数  
 default destructor 默认析构函数  
 delete operator delete 运算符  
 delete[] operator delete[] 运算符

dequeue (queue operation) 出队 (队列操作)  
 destructor 析构函数  
 dynamic objects 动态对象  
 enqueue (queue operation) 入队 (队列操作)  
 extensible language 可扩展语言  
 first-in-first-out (FIFO) 先进先出  
 forward class declaration 提前类声明  
 friend class 友元类  
 friend function 友元函数  
 host object 宿主对象  
 iterator 迭代器  
 last-in-first-out (LIFO) 后进先出  
 member selection operator (.) 成员选择运算符  
 member access specifiers 成员访问说明符  
 member initializer 成员初始化值  
 member object 成员对象

|                                          |             |                              |             |
|------------------------------------------|-------------|------------------------------|-------------|
| member object constructor                | 成员对象构造函数    | principle of least privilege | 最低权限原则      |
| new operator                             | new 运算符     | proxy class                  | 代理类         |
| new [] operator                          | new [] 运算符  | push ( stack operation )     | 压入 ( 堆栈操作 ) |
| object-based programming                 | 基于对象编程      | queue abstract data type     | 队列抽象数据类型    |
| operations in an ADT                     | ADT 中的操作    | stack abstract data type     | 堆栈抽象数据类型    |
| pointer member selection operator ( -> ) | 指针成员选择运算符   | static data member           | 静态数据成员      |
| pop ( stack operation )                  | 弹出 ( 堆栈操作 ) | static member function       | 静态成员函数      |
|                                          |             | this pointer                 | this 指针     |

## 自测练习

### 7.1 填空:

- \_\_\_\_\_ 语法用于初始化类的常量成员。
- 成员函数应声明为类的 \_\_\_\_\_ 才能访问这个类的 private 数据成员。
- \_\_\_\_\_ 运算符对指定类型对象动态分配内存并返回该类型的 \_\_\_\_\_。
- 常量对象应 \_\_\_\_\_, 不能在生成之后修改。
- \_\_\_\_\_ 数据成员表示类范围信息。
- 对象的成员函数能访问对象的“自我指针”, 称为 \_\_\_\_\_ 指针。
- 关键字 \_\_\_\_\_ 指定对象或变量初始化之后不可修改。
- 如果类的成员对象不提供成员初始化值, 则调用该对象的 \_\_\_\_\_。
- 如果成员函数不访问 \_\_\_\_\_ 类成员, 则可以声明为 static。
- 成员对象在所在类对象之 \_\_\_\_\_ 构造。
- \_\_\_\_\_ 运算符删除前面用 new 分配的内存。

### 7.2 找出下列各题的错误并说明如何纠正:

- ```

class Example {
public:
    Example( int y = 10 ) { data = y; }
    int getIncrementedData() const { return ++data; }
    static int getCount()
    {
        cout << "Data is " << data << endl;
        return count;
    }
private:
    int data;
    static int count;
}

```
- ```

char *string;
string = new char[ 20 ];
free( string );

```

## 自测练习答案

- 7.1 a) 成员初始化值。b) 友元。c) new、指针。d) 初始化。e) static。f) this。g) const。h) 默认构造函数。i) 非 static。j) 前。k) delete。
- 7.2 a) 不正确: Example 的类定义有两个错误。第一个在 getIncrementedData 函数中, 函数声

明为 const, 但其修改对象。

纠正: 要纠正第一个错, 删除 getIncrementedData 定义中的 const 关键字。

不正确: 第二个错在 getCount 函数中。函数声明为 static, 因此不能访问类的非 static 成员。

纠正: 要纠正第二个错, 删除 getCount 定义中的输出行。

b) 不正确: new 动态分配的内存用 C 标准库函数 free 删除。

纠正: 用 C++ 的 delete 运算符释放内存。C 语言式动态内存分配运算符不能与 C++ 的 new 和 delete 运算符混用。

## 练习

7.3 比较 C++ 的 new 和 delete 运算符动态内存分配与 C 标准库函数 malloc 和 free 运算符动态内存分配。

7.4 说明 C++ 友元关系的概念, 说明友元关系的副作用。

7.5 正确的 Time 类定义能否同时包括下列构造函数? 如果不能, 为什么?

```
Time ( int h = 0, int m = 0, int s = 0 );  
Time ();
```

7.6 构造函数或析构函数指定返回类型 (即使 void) 时, 会发生什么情况?

7.7 用下列条件生成 Date 类:

a) 用多种格式输出日期, 例如:

```
DDD YYYY  
MM/DD/YY  
June 14, 1992
```

b) 用重载的构造函数生成 Date 对象, 用 a) 中的日期格式初始化。

c) 生成一个 Date 构造函数, 用 time.h 头文件的标准库函数读取系统日期和设置 Date 成员。

第 8 章将介绍如何生成运算符, 测试两个日期的相等性和比较两个日期的前后顺序。

7.8 生成一个 SavingsAccount 类。用 static 数据成员包含每个存款人的 annualInterestRate (年利率)。类的每个成员包含一个 private 数据成员 savingsBalance, 表示当前存款额。提供一个 calculateMonthlyInterest 成员函数, 计算月利息, 用 balance 乘以 annualInterestRate 除以 12 取得, 并将这个月息加进 savingsBalance 中。提供一个 static 成员函数 modifyInterestRate, 将 static annualInterestRate 设置为新值。实例化两个不同的 SavingsAccount 对象 saver1 和 saver2, 结余分别为 2000.00 和 3000.00。将 annualInterestRate 设置为 3%, 计算每个存款人的月息并打印新的结果。然后将 annualInterestRate 设置为 4%, 再次计算每个存款人的月息并打印新的结果。

7.9 生成一个 IntegerSet 类。IntegerSet 类的每个对象可以保存 0 到 100 之间的整数值。一个集合内部表示为 0 和 1 的数组。数组元素 a[i] 为 1 表示整数 i 在集合中, 数组元素 a[j] 为 0 表示整数 j 不在集合中。默认构造函数将集合初始化为“空集”, 即所有元素都是 0。

提供常用集合操作的成员函数。例如, 提供一个 unionOfIntegerSets 成员函数, 生成两个现有集合的并集 (即只要其中一个集合的元素为 1, 则并集的元素就是 1, 如果两个集合的元素均为 0, 则并集的元素就是 0)。

提供 `intersectionOfIntegerSets` 成员函数生成两个现有集合的交集(即只要其中一个集合的元素为 0, 交集的元素就是 0, 如果两个集合的元素均为 1, 则交集的元素就是 1)。

提供一个 `insertElement` 成员函数, 在集合中插入新整数 `k` (将 `a[k]` 设置为 1)。提供 `deleteElement` 删除整数 `m` (将 `a[m]` 设置为 0)。

提供一个 `setPrint` 成员函数, 将集合表示的值打印为以空格分隔的列表。只打印集合中存在的元素(即对应值为 1 的位置), 空集打印 ---。

提供一个 `isEqualTo` 成员函数, 确定两个集合是否相等。

提供其他构造函数, 取五个整数参数, 可以初始化一组对象。如果提供的元素不到五个, 其他元素用默认参数 -1。

编写一个驱动程序, 测试 `IntegerSet` 类。实例化几个 `IntegerSet` 对象。测试所有成员函数能否正确工作。

- 7.10 图 7.8 的 `Time` 类可以在内部将时间表示为从午夜算起的秒数而不是三个整数值 `hour`、`minute` 和 `second`。客户可以用相同的 `public` 方法并取得相同结果。修改图 7.8 的 `Time` 类, 将时间表示为从午夜算起的秒数, 证明类的客户看不到功能性的变化。

## 第8章 运算符重载

### 教学目标

- 了解如何重新定义（重载）运算符以处理新类型
- 了解如何将一个类的对象转换为另一个类的对象
- 了解重载运算符的时机
- 学习几个使用运算符重载的例子
- 生成 Array、String 和 Date 类

### 8.1 简介

第6章和第7章介绍了C++类的基本知识和抽象数据类型的表示方法。对类的对象（即抽象数据类型的实例）的操作是通过向对象发送消息完成的（即调用成员函数的形式）。对某些类（特别是数学类）来说，这种调用方式是繁琐的，而用C++中的丰富的内部运算符集来指定对对象的操作要更好。本章要介绍怎样把C++中的运算符和类的对象结合在一起使用，这个过程称为运算符重载。扩展C++使它具有这些新的功能是理所当然的。

运算符<<在C++中有多种用途，既可以用作流插入运算符又可以用作左移位运算符，这是运算符重载的一个范例。同样，运算符>>也是C++中的一个重载运算符，它既可以用作流读取运算符，也可以用作右移位运算符。这两个运算符都是在C++类库中重载的。C++语言本身也重载了运算符+和-，这两个运算符在整数算术运算、浮点数算术运算和指针算术运算等上下文中执行的操作是不同的。

为了使运算符在不同的上下文中具有不同的含义，C++允许程序员重载大多数运算符。编译器根据运算符的使用方式产生合适的代码。某些运算符（特别是赋值运算符以及+和-等等的各种算术运算符）经常要被重载。虽然重载运算符所能够实现的任务也能够用明确的函数调用完成，但是使用重载运算符能够使程序更易于阅读。

本章要讨论使用运算符重载的时机以及怎样重载运算符，还要介绍使用重载运算符的许多完整程序。

### 8.2 运算符重载的基础

C++程序设计是对类型敏感的，并且程序设计的重点也是放在类型上。程序员可使用内部的类型，也可以定义新的类型。内部的类型可以和C++中丰富的运算符集一起使用。运算符为程序员提供了操作内部类型对象的简洁的表示方法。

程序员也可以把运算符和用户自定义的类型一起使用。尽管C++不允许建立新的运算符,但是允许重载现有的运算符,使它在用于类的对象时具有新类型的含义,这是C++最强大的特点之一。

#### 软件工程视点 8.1

运算符重载提供了C++的可扩展性,这也是C++最吸引人的属性之一。

#### 编程技巧 8.1

在完成同样的操作的情况下,如果运算符重载能够比用明确的函数调用使程序更清晰,则应该使用运算符重载。

#### 编程技巧 8.2

不要过度地或不合理地使用运算符重载,因为这样会使程序语义不清且难以阅读。

虽然运算符重载听起来好像是C++的外部能力,但是多数程序员都不知不觉地使用过重载的运算符。例如,加法运算符(+)对整数、单精度数和双精度数的操作是大不相同的。但是,因为C++语言本身已经重载了该运算符,所以它能够用于int、float、double和其他内部定义类型的变量。

运算符重载是通过编写函数定义实现的。函数定义虽然也包括函数首部和函数体,但是函数名是由关键字operator和其后要重载的运算符符号组成的。例如,函数名operator+重载了运算符+。

用于类的对象的运算符必须重载,但是有两种例外情况。赋值运算符(=)无需重载就可用于每一个类。在不提供重载的赋值运算符时,赋值运算符的默认行为是复制类的数据成员。不久就会看到,这种默认的复制行为对于带有指针成员类是危险的,对这种类通常要显式重载赋值运算符。地址运算符&也无需重载就可以用于任何类的对象,它返回对象在内存中的地址。地址运算符也可以被重载。

运算符重载最适合用于数学类。为了与在现实世界中操作这些数学类的方式一致,通常要重载一组运算符。例如,对于复数类,通常不仅仅要重载运算符+,因为其他算术运算符也经常用于复数。

C++语言的运算符很丰富。因为程序员对每个运算符的含义和使用的具体语境是理解的,所以在重载用于新类的运算符时,程序员能够根据运算符的意义做出合理的选择。

C++为其内部类型提供了丰富的运算符集,重载这些运算符的目的是为用户自定义的类型提供同样简洁的表达式。然而,运算符的重载不是自动完成的,程序员必须为所要执行的操作编写运算符重载函数。有时最好把这些函数用作成员函数,有时最好用作友元函数,在极少数情况下,他们可能既不是成员函数,也不是友元函数。

可能会发生重载误用的情况,例如重载加法运算符(+)使它执行类似于减法的运算,或者重载除法运算符(/)以使它执行类似于乘法的运算。如此使用重载会使程序令人迷惑不解。

#### 编程技巧 8.3

在把重载运算符用于类的对象时,重载运算符的功能类似于该运算符作用于内部类型的对象时所完成的功能,避免没有目的地使用重载运算符。

#### 编程技巧 8.4

在用重载运算符编写C++程序之前,查阅编译器的手册,了解特定运算符的各种限制和要求。

## 8.3 运算符重载的限制

C++ 中的大部分运算符都可以被重载。图 8.1 列出了可以被重载的运算符，图 8.2 列出了不能被重载的运算符。

### 常见编程错误 8.1

想重载不能重载的运算符是个语法错误。

| 可以被重载的运算符 |           |    |    |    |    |     |        |
|-----------|-----------|----|----|----|----|-----|--------|
| +         | -         | *  | /  | %  | ^  | &   |        |
| ~         | !         | =  | <  | >  | += | --  | *=     |
| /=        | %=        | ^= | &= | =  | << | >>  | >>=    |
| <<=       | ==        | != | <= | >= | && |     | ++     |
| --        | ->*       | '  | -> | [] | () | new | delete |
| new []    | delete [] |    |    |    |    |     |        |

图 8.1 可以被重载的运算符

| 不可以被重载的运算符 |    |    |        |
|------------|----|----|--------|
| .          | .* | :: | ? :    |
|            |    |    | sizeof |

图 8.2 不能被重载的运算符

重载不能改变运算符的优先级。虽然重载具有固定优先级的运算符可能会不便于使用，但是在表达式中使用圆括号可以强制改变重载运算符的计算顺序。

重载不能改变运算符的结合律。

重载不能改变运算符操作数的个数。重载的一元运算符仍然是一元运算符，重载的二元运算符仍然是二元运算符，C++ 中的惟一的三元运算符 (?:) 也不能被重载。运算符 &、\*、+ 和 - 既可以用作一元运算符，也可以用作二元运算符，可以分别把他们重载为一元运算符和二元运算符。

不能创建新的运算符，只有现有的运算符才能被重载。因此，程序员不能使用一些流行的表示方法，如 BASIC 中表示指数的运算符 \*\*。

### 常见编程错误 8.2

试图创建新的运算符是个语法错误。

运算符重载不能改变该运算符用于内部类型对象时的含义。例如，程序员不能改变运算符 + 用于两个整数时的含义。运算符重载只能和用户自定义类型的对象一起使用，或者用于用户自定义类型的对象和内部类型的对象混合使用时。

### 常见编程错误 8.3

试图改变运算符对内部类型的对象的作用方式是个语法错误。

### 软件工程视点 8.2

运算符函数的参数至少有一个必须是类的对象或者是对类的对象的引用。这种规定防止了程序员改变运算符对内部类型的对象的作用方式。

重载了赋值运算符 = 和加法运算符 + 以后，虽然下列语句是允许的：



```
object2 = object2 + object1;
```

但并不意味运算符 += 也被自动重载了。因此,下面的语句是不允许的:

```
object2 += object1;
```

然而,显式地重载运算符 += 可使上述语句成立。

#### 常见编程错误 8.4

认为重载了某个运算符(如“+”)可以自动地重载相关的运算符(如“+=”),或重载了“==”就自动重载了“!=”,运算符只能被显式重载(不存在隐式重载)。

#### 常见编程错误 8.5

想通过运算符重载改变运算符的“数量”是个语法错误。

#### 编程技巧 8.5

要保证相关运算符的一致性,可以用一个运算符实现另一个运算符(即用重载的运算符“+”实现重载的运算符“+=”)。

## 8.4 用作类成员与友元函数的运算符函数

运算符函数既可以是成员函数,也可以是非成员函数。非成员函数通常是友元函数。成员函数是用 this 指针隐式地访问类对象的某个参数,非成员函数的调用必须明确地列出该参数。

在重载运算符()、[]、~>或者任何赋值运算符时,运算符重载函数必须声明为类的一个成员。对于其他的运算符,运算符重载函数可以是非成员函数。

不管运算符函数是成员函数还是非成员函数,运算符在表达式中的使用方式是相同的。哪种实现方式更好呢?

当运算符函数是一个成员函数时,最左边的操作数(或者只有最左边的操作数)必须是运算符类的一个类对象(或者是对该类对象的引用)。如果左边的操作数必须是一个不同类的对象,或者是一个内部类型的对象,该运算符函数必须作为一个非成员函数来实现(正如8.5节中分别重载运算符<<和>>作为流插入运算符和流读取运算符一样)。运算符函数作为非成员函数直接访问该类的private或者protected成员时,该函数必须是一个友元。

重载的<<运算符必须有一个类型为 ostream & 的左操作数(例如表达式 cout<<classObject 中的 cout),因此它必须是一个非成员函数。类似地,重载 >> 运算符必须有一个类型为 istream & 的左操作数(如表达式 cin >> classObject 中的 cin),所以它也必须是一个非成员函数。此外,这两个重载的运算符函数都需要访问输出或输入的类对象的private数据成员,因此出于性能考虑,这些重载的运算符函数通常都是类的友元函数。

#### 性能提示 8.1

可以把一个运算符作为一个非成员、非友元函数重载。但是,这样的运算符函数访问类的private和protected数据时必须使用类的public接口中提供的“set”或者“get”函数(即设置数据和读取数据的函数),调用这些函数的开销会降低性能,因此必须内联这些函数以提高性能。

只有当二元运算符的最左边的操作数是该类的一个对象时,或者当一元运算符的操作数是该类的一个对象时,才需调用特定类的运算符成员函数。

选择非成员函数重载运算符的另外一个原因是使运算符具有可交换性。例如：假定有 long int 类型的一个对象 number 和类 HugeInteger 的一个对象 bigInteger1 (本章的练习中开发了类 HugeInteger, 该类中的整数可以是任意大小, 不受机器字长的限制)。如果要求加法运算符 (+) 生成一个临时的 HugeInteger 对象, 它是 HugeInteger 和 long int 类型对象的和 (如表达式 bigInteger1 + number), 或者是 long int 和 HugeInteger 类型对象的和 (如表达式 number + bigInteger1), 那么上述的加法运算符就要具有可交换性 (正如通常的加法一样)。问题在于, 如果把运算符作为成员函数重载, 类的对象必须出现在运算符的左边, 所以要将运算符函数作为一个非成员的友元重载, 这样才能允许 HugeInteger 对象出现在加法运算符的右边。处理 HugeInteger 对象在左边的 operator+ 函数依然可以是一个成员函数。记住, 非成员函数不一定是友元, 只要类的 public 接口中有相应 set 和 get 函数, 有内联的 set 和 get 函数则更好。

## 8.5 重载流插入与流读取运算符

C++ 的流读取运算符 >> 和流插入运算符 << 可用来输入输出标准类型的数据。这两个运算符是 C++ 编译器在类库中提供的, 可以处理包括类 C 语言中的 char\* 字符串和指针在内的每一种内部数据类型。也可以重载这两个运算符以输入输出用户自定义类型的数据。图 8.3 中的程序演示了重载的流读取运算符和流插入运算符, 它们用来处理用户自定义的电话号码类 PhoneNumber 的数据。程序假定输入的电话号码是正确的, 错误检测留给读者在练习中完成。

```

1 // Fig. 8.3: fig08_03.cpp
2 // Overloading the stream-insertion and
3 // stream-extraction operators.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 class PhoneNumber {
8     friend ostream &operator<< (ostream&, const PhoneNumber &);
9     friend istream &operator>> (istream&, PhoneNumber &);
10
11 private:
12     char areaCode[ 4 ]; // 3-digit area code and null
13     char exchange[ 4 ]; // 3-digit exchange and null
14     char line[ 5 ]; // 4-digit line and null
15 };
16
17 // Overloaded stream-insertion operator (cannot be
18 // a member function if we would like to invoke it with
19 // cout << somePhoneNumber;).
20 ostream &operator<< (ostream &output, const PhoneNumber &num)
21 {
22     output << "(" << num.areaCode << " ) "
23         << num.exchange << "-" << num.line;
24     return output; // enables cout << a << b << c;
25 }
26
27 istream &operator>> (istream &input, PhoneNumber &num)
28 {
29     input.ignore(); // skip (
30     input >> setw( 4 ) >> num.areaCode; // input area code
31     input.ignore( 2 ); // skip ) and space

```

```

32  input >> setw( 4 ) >> num.exchange; // input exchange
33  input.ignore();                      // skip dash (-)
34  input >> setw( 5 ) >> num.line;      // input line
35  return input;                        // enables cin >> a >> b >> c;
36 }
37
38 int main()
39 {
40     PhoneNumber phone; // create object phone
41
42     cout << "Enter phone number in the form (123) 456-7890:\n";
43
44     // cin >> phone invokes operator>> function by
45     // issuing the call operator>>( cin, phone ).
46     cin >> phone;
47
48     // cout << phone invokes operator<< function by
49     // issuing the call operator<<( cout, phone ).
50     cout << "The phone number entered was: " << phone << endl;
51     return 0;
52 }

```

Enter phone number in the form (123) 456-7890:  
(800) 555-1212  
The phone number entered was: (800) 555-1212

图 8.3 用户自定义的流插入和流读取运算符

流读取运算符函数 `operator >>` (第 27 行) 含有两个参数, 一个是对 `istream` 的引用 (即程序中的 `input`), 另一个则是对用户自定义类型 `PhoneNumber` 的引用 (即程序中的 `num`)。函数返回一个对 `istream` 的引用。在图 8.3 的程序中, 运算符函数 `operator >>` 用来把下述格式的电话号码输入到类 `PhoneNumber` 的对象中:

```
(800) 555-1212
```

当编译器遇到 `main()` 函数中的表达式:

```
cin >> phone
```

编译器将生成函数调用:

```
operator >> (cin, phone);
```

当执行该调用时, 引用参数 `input` 成为 `cin` 的一个别名, `num` 成为 `phone` 的一个别名。运算符函数使用 `istream` 的成员函数 `getline`, 将电话号码的三部分作为字符串分别读到被引用的 `PhoneNumber` 对象 (运算符函数中的 `num` 和 `main` 函数中的 `phone`) 的 `areaCode`、`exchange` 和 `line` 成员中。流操纵算子 `setw` 保证将正确的字符数读入到字符数组中。回忆一下, 我们曾经使用 `cin` 和 `setw` 限制读入的字符数比参数少 1 (例如 `setw(4)` 只允许读入 3 个字符, 留出一个位置保存 `null` 终止符)。通过调用 `istream` 的成员函数 `ignore` 跳过括号、空格、破折号等等 (`ignore` 函数删除输入流中指定数目的字符, 默认个数为 1)。函数 `operator >>` 返回对 `istream` 对象的引用 `input` (即 `cin`), 因而能够在 `PhoneNumber` 对象的输入操作完成后, 继续执行对 `PhoneNumber` 的其他对象或者其他数据类型对象的输入操作。例如, 可以像下面那样输入两个 `PhoneNumber` 对象:

```
cin >> phone1 >> phone2;
```

首先是表达式 `cin >> phone1` 产生如下调用:

```
operator >> (cin, phone1);
```

该调用返回 `cin`, 并把它作为 `cin >> phone1` 的值, 因此表达式的其余部分将被简单地解释为 `cin >> phone2`, 这将通过下列调用执行:

```
operator >> (cin, phone2);
```

流插入运算符有两个参数, 一个是对 `ostream` 的引用 (即 `output`), 另一个是对用户自定义类型 `PhoneNumber` 的引用 (即 `num`), 函数返回一个对 `ostream` 的引用。函数 `operator<<` 显示了 `PhoneNumber` 的对象。当编译器遇到 `main` 函数中的表达式:

```
cout << phone
```

编译器生成非成员函数调用:

```
operator << (cout, phone);
```

因为电话号码的各个部分是以字符串的格式存储的, 所以函数 `operator<<` 以字符串形式显示它们。

注意, 函数 `operator<<` 和 `operator>>` 在类 `PhoneNumber` 中被声明为友元函数而不是成员函数。因为要把类 `PhoneNumber` 的对象作为运算符的右操作数, 所以这些运算符函数必须是非成员函数。要把运算符重载为成员函数, 类的操作数 (类的对象) 必须出现在运算符的左边。如果重载的输入和输出运算符必须直接访问类的非 `public` 成员, 则必须把它们声明为友元。另外, 还要注意 `operator<<` 参数表中引用的 `PhoneNumber` 是 `const` 类型 (因为只输出 `PhoneNumber`), 而 `operator>>` 参数表中引用的 `PhoneNumber` 是非 `const` 类型 (由于 `PhoneNumber` 对象要修改成在该对象中存放输入的电话号码)。

### 软件工程视点 8.3

无需修改类 `ostream` 和 `istream` 的声明和 `private` 数据成员就可以给用户自定义类型添加新的输入/输出能力。这种方式提高了 C++ 语言的可扩展性, 可扩展性是 C++ 的最具吸引力的特点。

## 8.6 重载一元运算符

类的一元运算符可重载为一个没有参数的非 `static` 成员函数或者带有一个参数的非成员函数, 参数必须是用户自定义类型的对象或者对该对象的引用。实现重载运算符的成员函数应是非 `static`, 以便访问类的非 `static` 数据。记住, `static` 成员函数只能访问类的 `static` 数据成员。

本章稍后要用重载的一元运算符 “!” 测试一个字符串是否为空并返回一个布尔值。当把一元运算符 (如 “!”) 重载为没有参数的非 `static` 成员函数时, 如果 `s` 是 `String` 类的对象或是对 `String` 类对象的引用, 那么编译器在遇到表达式 `!s` 时会生成函数调用 `s.operator!()`。操作数 `s` 是类的对象, 它调用了 `String` 类的成员函数 `operator!`。类定义中的函数声明如下:

```
class String {  
public:  
    bool operator!() const;  
    ...  
};
```

把一元运算符（如“!”）重载为带有一个参数的非成员函数时，参数有两种不同的情况。一种情况是该参数是某个对象（需要对象的副本，因此函数不作用于原对象），另一种情况是该参数是对某个对象的引用（不复制原对象，因此函数会作用于原对象）。如果参数 *s* 是 *String* 类的一个对象或对 *String* 类对象的引用，则 *!s* 将被处理为 *operator!(s)*，调用 *String* 类的非成员友元函数。*String* 类声明如下：

```
class String {
    friend bool operator!( const String & );
    ...
};
```

#### 编程技巧 8.6

重载一元运算符时，把运算符函数用作类的成员而不用作友元函数。因为友元的使用破坏了类的封装，所以除非绝对必要，否则应尽量避免使用友元函数和友元类。

## 8.7 重载二元运算符

二元运算符可以重载为带有一个参数的非 *static* 成员函数，或者带有两个参数的非成员函数（参数之一必须是类的对象或者是对类的对象的引用）。

本章稍后要重载运算符 *+=*，当把它重载为带有一个参数的 *String* 类的非 *static* 成员函数时，如果 *y* 和 *z* 是 *String* 类的对象，则 *y += z* 将被处理为 *y.operator+=( z)*，调用成员函数 *operator+=*，声明如下：

```
class String {
public:
    const String &operator+=( const String & );
    ...
};
```

二元运算符 *+=* 也可以重载为带有两个参数的非成员函数，其中的一个参数必须是类的对象或者是对类的对象的引用。如果 *y* 和 *z* 是 *String* 类的对象，则 *y += z* 将被处理为 *operator+=(y, z)*，调用友元函数 *operator+=*，声明如下：

```
class String {
    friend const String &operator+=( String &,
                                    const String & );
    ...
};
```

## 8.8 实例研究：Array 类

在 C 和 C++ 中，数组是一种指针，因而数组存在许多导致错误的陷阱。例如，由于 C 和 C++ 不检测下标是否超出数组的边界而使程序导致越界错误；大小为 *n* 的数组的下标必须是 0、1、2、...、*n*-1，下标是不允许改变的；不能一次输入或输出整个数组，而只能单独读取或者输出每个数组元素；不能用相等运算符或者关系运算符比较两个数组（因为数组名仅仅是指向内存中数组起始位置的指针）；当把一个数组传递给一个能处理任意大小数组的常用函数时，数组的大小也必须作为一

个额外的参数传递给该函数；不能用赋值运算符把一个数组赋给另一个数组（因为数组名是 `const` 类型指针，而常量指针不能用于赋值运算符的左边）。尽管所有这些处理能力似乎应该是很自然的，但是 C 和 C++ 都没有提供这种能力。然而，C++ 提供了实现这种能力的手段，这就是运算符重载。

本节的范例建立了一个数组类，它能检测范围以确保数组下标不会越界，允许用赋值运算符把一个数组赋给另外一个数组。数组对象自动知道数组的大小，因而不用将数组的大小传送给函数。可以用流读取运算符和流插入运算符输入输出整个数组。还可以用相等运算符 `==` 和 `!=` 比较数组。范例程序中的数组类用一个 `static` 成员跟踪程序中实例化数组对象的数目。

本例将加深读者对数据抽象的认识。当然，读者还可以增加数组类的其他功能，类的开发是十分有趣并富有挑战性的。

图 8.4 中的程序演示了类 `Array` 和用于该类的重载运算符。首先来看一下 `main` 函数中的驱动程序，然后再探讨类的定义以及类的每个成员和友元函数的定义。

```

1 // Fig. 8.4: array1.h
2 // Simple class Array (for integers)
3 #ifndef ARRAY1_H
4 #define ARRAY1_H
5
6 #include <iostream.h>
7
8 class Array {
9     friend ostream &operator<<( ostream &, const Array & );
10    friend istream &operator>>( istream &, Array & );
11 public:
12    Array( int = 10 );           // default constructor
13    Array( const Array & );      // copy constructor
14    ~Array();                   // destructor
15    int getSize() const;         // return size
16    const Array &operator=( const Array & ); // assign arrays
17    bool operator==( const Array & ) const; // compare equal
18
19    // Determine if two arrays are not equal and
20    // return true, otherwise return false (uses operator==).
21    bool operator!=( const Array &right ) const
22    { return ! ( *this == right ); }
23
24    int &operator[] ( int );      // subscript operator
25    const int &operator[] ( int ) const; // subscript operator
26    static int getArrayCount();   // Return count of
27                                // arrays instantiated.
28 private:
29    int size; // size of the array
30    int *ptr; // pointer to first element of array
31    static int arrayCount; // # of Arrays instantiated
32 };
33
34 #endif
35 // Fig 8.4: array1.cpp
36 // Member function definitions for class Array
37 #include <iostream.h>
38 #include <iomanip.h>
39 #include <stdlib.h>

```

```
40 #include <assert.h>
41 #include "array1.h"
42
43 // Initialize static data member at file scope
44 int Array::arrayCount = 0; // no objects yet
45
46 // Default constructor for class Array (default size 10)
47 Array::Array( int arraySize )
48 {
49     size = ( arraySize > 0 ? arraySize : 10 );
50     ptr = new int[ size ]; // create space for array
51     assert( ptr != 0 ); // terminate if memory not allocated
52     ++arrayCount; // count one more object
53
54     for ( int i = 0; i < size; i++ )
55         ptr[ i ] = 0; // initialize array
56 }
57
58 // Copy constructor for class Array
59 // must receive a reference to prevent infinite recursion
60 Array::Array( const Array &init ) : size( init.size )
61 {
62     ptr = new int[ size ]; // create space for array
63     assert( ptr != 0 ); // terminate if memory not allocated
64     ++arrayCount; // count one more object
65
66     for ( int i = 0; i < size; i++ )
67         ptr[ i ] = init.ptr[ i ]; // copy init into object
68 }
69
70 // Destructor for class Array
71 Array::~Array()
72 {
73     delete [] ptr; // reclaim space for array
74     --arrayCount; // one fewer objects
75 }
76
77 // Get the size of the array
78 int Array::getSize() const { return size; }
79
80 // Overloaded assignment operator
81 // const return avoids: ( a1 = a2 ) = a3
82 const Array &Array::operator=( const Array &right )
83 {
84     if ( &right != this ) { // check for self-assignment
85
86         // for arrays of different sizes, deallocate original
87         // left side array, then allocate new left side array.
88         if ( size != right.size ) {
89             delete [] ptr; // reclaim space
90             size = right.size; // resize this object
91             ptr = new int[ size ]; // create space for array copy
92             assert( ptr != 0 ); // terminate if not allocated
93         }
94     }
```

```
95     for ( int i = 0; i < size; i++ )
96         ptr[ i ] = right.ptr[ i ]; // copy array into object
97     }
98
99     return *this;    // enables x = y = z;
100 }
101
102 // Determine if two arrays are equal and
103 // return true, otherwise return false.
104 bool Array::operator==( const Array &right ) const
105 {
106     if ( size != right.size )
107         return false;    // arrays of different sizes
108
109     for ( int i = 0; i < size; i++ )
110         if ( ptr[ i ] != right.ptr[ i ] )
111             return false; // arrays are not equal
112
113     return true;        // arrays are equal
114 }
115
116 // Overloaded subscript operator for non-const Arrays
117 // reference return creates an lvalue
118 int &Array::operator[]( int subscript )
119 {
120     // check for subscript out of range error
121     assert( 0 <= subscript && subscript < size );
122
123     return ptr[ subscript ]; // reference return
124 }
125
126 // Overloaded subscript operator for const Arrays
127 // const reference return creates an rvalue
128 const int &Array::operator[]( int subscript ) const
129 {
130     // check for subscript out of range error
131     assert( 0 <= subscript && subscript < size );
132
133     return ptr[ subscript ]; // const reference return
134 }
135
136 // Return the number of Array objects instantiated
137 // static functions cannot be const
138 int Array::getArrayCount() { return arrayCount; }
139
140 // Overloaded input operator for class Array;
141 // inputs values for entire array.
142 istream &operator>>( istream &input, Array &a )
143 {
144     for ( int i = 0; i < a.size; i++ )
145         input >> a.ptr[ i ];
146
147     return input;    // enables cin >> x >> y;
148 }
149
150 // Overloaded output operator for class Array
```



```
151 ostream &operator<<( ostream &output, const Array &a )
152 {
153     int i;
154
155     for ( i = 0; i < a.size; i++ ) {
156         output << setw( 12 ) << a.ptr[ i ];
157
158         if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
159             output << endl;
160     }
161
162     if ( i % 4 != 0 )
163         output << endl;
164
165     return output;    // enables cout << x << y;
166 }
167 // Fig. 8.4: fig08_04.cpp
168 // Driver for simple class Array
169 #include <iostream.h>
170 #include "array1.h"
171
172 int main()
173 {
174     // no objects yet
175     cout << "# of arrays instantiated = "
176          << Array::getArrayCount() << '\n';
177
178     // create two arrays and print Array count
179     Array integers1( 7 ), integers2;
180     cout << "# of arrays instantiated = "
181          << Array::getArrayCount() << "\n\n";
182
183     // print integers1 size and contents
184     cout << "Size of array integers1 is "
185          << integers1.getSize()
186          << "\nArray after initialization:\n"
187          << integers1 << '\n';
188
189     // print integers2 size and contents
190     cout << "Size of array integers2 is "
191          << integers2.getSize()
192          << "\nArray after initialization:\n"
193          << integers2 << '\n';
194
195     // input and print integers1 and integers2
196     cout << "Input 17 integers:\n";
197     cin >> integers1 >> integers2;
198     cout << "After input, the arrays contain:\n"
199          << "integers1:\n" << integers1
200          << "integers2:\n" << integers2 << '\n';
201
202     // use overloaded inequality (!=) operator
203     cout << "Evaluating: integers1 != integers2\n";
204     if ( integers1 != integers2 )
205         cout << "They are not equal\n";
206 }
```

```

207 // create array integers3 using integers1 as an
208 // initializer; print size and contents
209 Array integers3( integers1 );
210
211 cout << "\nSize of array integers3 is "
212      << integers3.getSize()
213      << "\nArray after initialization:\n"
214      << integers3 << '\n';
215
216 // use overloaded assignment (=) operator
217 cout << "Assigning integers2 to integers1:\n";
218 integers1 = integers2;
219 cout << "integers1:\n" << integers1
220      << "integers2:\n" << integers2 << '\n';
221
222 // use overloaded equality (==) operator
223 cout << "Evaluating: integers1 == integers2\n";
224 if ( integers1 == integers2 )
225     cout << "They are equal\n\n";
226
227 // use overloaded subscript operator to create rvalue
228 cout << "integers1[ 5] is " << integers1[ 5] << '\n';
229
230 // use overloaded subscript operator to create lvalue
231 cout << "Assigning 1000 to integers1[ 5]\n";
232 integers1[ 5] = 1000;
233 cout << "integers1:\n" << integers1 << '\n';
234
235 // attempt to use out of range subscript
236 cout << "Attempt to assign 1000 to integers1[ 15]" << endl;
237 integers1[ 15] = 1000; // ERROR: out of range
238
239 return 0;
240 }

```

**输出结果:**

```

# of arrays instantiated = 0
# of arrays instantiated = 2

```

```

Size of array integers1 is 7
Array after initialization:

```

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 |   |

```

Size of array integers2 is 10
Array after initialization:

```

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 |   |   |

```

Input 17 integers:

```

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

```

```

After input, the arrays contain:

```

```

integers1:
      1      2      3      4
      5      6      7

```

```

integers2:
    8          9          10         11
    12         13         14         15
    16         17

Evaluating: integers1 != integers2
They are not equal

Size of array integers3 is 7
Array after initialization:
    1          2          3          4
    5          6          7

Assigning integers2 to integers1:
integers1:
    8          9          10         11
    12         13         14         15
    16         17
integers2:
    8          9          10         11
    12         13         14         15
    16         17

Evaluating: integers1 == integers2
They are equal

integers1[ 5] is = 13
Assigning 1000 to integers1[ 5]
integers1:
    8          9          10         11
    12         1000       14         15
    16         17

Attempt to assign 1000 to integers1[ 15]
Assertion failed: 0 <= subscript && subscript < size,
file Array1.cpp, line 87

abnormal program termination

```

图 8.4 用重载运算符演示 Array 类

类 Array 的 static 类变量 arrayCount 包含了程序执行过程中实例化的 Array 对象的个数, 该值由第 176 行的 static 成员函数 getArrayCount 返回。程序实例化了类 Array 的两个对象 (第 179 行), 对象 integers1 有 7 个元素, 对象 integers2 有 10 个元素 (默认的元素个数由类 Array 的构造函数指定)。第 181 行再次调用 getArrayCount, 取得类变量 arrayCount 的值。第 184 到第 187 行用成员函数 getSize 确定 Array integers1 的长度, 并使用 Array 重载流插入运算符输出 integers1, 以证实构造函数正确地初始化了数组的元素。接下来, 第 190 到第 193 行的程序先输出数组 integers2 的长度, 然后用重载的流插入运算符输出 integers2 数组。

完成上述工作后, 程序提示用户输入 17 个整数, Array 重载流读取运算符并使用下列语句 (第 197 行):

```
cin >> integers1 >> integers2;
```

把这些值读入到两个数组中，前7个整数保存在 `integers1` 中，其余的则保存在 `integers2` 中。为了证实输入操作的正确性，程序用流插入运算符输出了这两个数组（第198到第200行）。

接下来，程序通过测试条件（第204行）：

```
integers1 != integers2
```

来验证重载的不相等运算符 `!=`，输出结果表明这两个数组确实不相等。

第209行程序实例化第三个数组 `integers3` 并用数组 `integers1` 对其初始化，这将调用 `Array` 复制构造函数将 `integers1` 复制到 `integers3` 中。我们将在后面详细讨论复制构造函数。

程序在第211到第214行输出 `integers3` 的长度，并使用 `Array` 重载流插入运算符输出 `integers3`，以证实构造函数正确地初始化数组。

接下来，第218行通过下列语句测试重载的赋值运算符（`=`）：

```
integers1 = integers2;
```

然后打印出这两个数组来验证赋值的正确性。原来的 `integers1` 中只有7个整数，现在必须要使其能容纳 `integers2` 中的10个元素的副本。重载的赋值运算符可以改变原先的 `integers1` 的大小并复制 `integers2` 中的元素。

接下来，第224行用重载的运算符 `==` 测试赋值后 `integers1` 和 `integers2` 是否相等。

接下来，第228行用重载的下标运算符引用 `integers1[5]`（`integers1` 数组范围内的一个元素），这个带下标的数组名作为右值来打印 `integers1[5]` 的值，第232行将 `integers[5]` 放在赋值语句左边并赋给其一个新值1000，注意，`operator[]` 返回引用并作为左值使用（在确定了5是在 `integers1` 的长度范围内）。

第237行程序试图将1000赋给 `integers[15]`（越界的元素）。`Array` 重载了 `[]` 运算符捕获到该错误并中止程序。

有意思的是，数组下标运算符不仅仅可用于数组，还可以用于从其他各种容器类（如链表、字符串、字典等等）中选择元素。此外，下标不仅仅是整数，还可以是字符、字符串、浮点数甚至是用户自定义的对象。

上面介绍了程序是如何执行的。下面再分析一下类的首部和成员函数的定义。第29行到第31行：

```
int size; // size of the array
int *ptr; // pointer to first element of array
static int arrayCount; // # of Arrays instantiated
```

是类的 `private` 数据成员，包括一个 `int` 类型指针 `ptr`（指向 `Array` 对象中存储整型的动态分配数组）、一个表示数组元素个数的 `size` 成员以及一个表示已经实例化的数组对象数目的 `static` 成员 `arrayCount`。

第9行和第10行：

```
friend ostream &operator<<( ostream &, const Array & );
friend istream &operator<<( istream &, Array & );
```

声明了重载的流插入、流读取运算符是类 `Array` 的友元。当编译器遇到表达式：

```
cout << arrayObject
```

通过生成 `operator<<(cout, arrayObject)` 来调用函数 `operator<<(ostream &, const Array &)`。当编译器遇到表达式：

```
cin >> arrayObject
```

通过生成 `operator>>(cin, arrayObject)` 来调用函数 `operator>>(istream &, Array &)`。

因为 `Array` 对象总是在流插入运算符和流读取运算符的右边，所以这两个运算符函数不能是 `Array` 的成员函数。如果这些运算符函数是 `Array` 的成员函数，则可以用下列的语句（可能会出现意外情况）输入和输出 `Array`：

```
arrayObject << cout;
arrayObject >> cin;
```

函数 `operator<<`（在第 151 行定义）打印由 `size` 指定的存储在 `ptr` 中的数组元素的个数，而函数 `operator>>`（在第 142 行定义）则把数据直接输入到 `ptr` 所指向的数组中。为了能够分别实现连续的输入输出，这两个运算符都返回了一个合适的引用。

代码行：

```
Array( int = 10 ); // default constructor
```

声明了类的默认构造函数，并且指定数组元素的默认大小为 10。当编译器遇到如下声明：

```
Array integers1( 7 );
```

或与之等价的形式：

```
Array integers1 = 7;
```

编译器将调用默认构造函数（本例中默认构造函数实际上接收一个 `int` 参数，默认值为 10）。默认构造函数（第 47 行定义）验证参数并赋值给 `size` 数据成员，用 `new` 分配数组所需的内存，将 `new` 返回的指针赋给数据成员 `ptr`，然后用 `assert` 测试 `new` 操作是否成功，并递增 `arrayCount` 的值，最后用 `for` 循环将数组的所有元素初始化为 0。如果没有将 `Array` 初始化，也可以在以后读取相应的值，但这样做会降低程序的可执行性。`Array` 和任何对象都应随时保持正确初始化和一致的状态。

第 13 行：

```
Array (const Array &); // copy constructor
```

是一个复制构造函数（第 60 行定义），它通过建立现有 `Array` 对象的副本来初始化 `Array` 对象。必须要小心对待这种复制操作，避免两个 `Array` 对象指向同一块动态分配的存储区，默认的成员复制更容易发生这种问题。不论何时需要复制对象时都会调用复制构造函数，如在传值调用时、从被调用函数返回一个对象时、或把某个对象初始化为同类的另外一个对象的副本时。当声明创建类 `Array` 的一个对象并用另外一个对象对它初始化时，调用复制构造函数。例如下列声明：

```
Array integers3( integers1 );
```

或者与之等价的声明：

```
Array integers3 = integers1;
```

#### 常见编程错误 8.6

注意复制构造函数要按引用调用，而不是按值调用，否则复制构造函数调用会造成无穷递归（这是个致命逻辑错误），因为对于按值调用，建立传入复制构造函数的对象副本会造成复制构造函数的递归调用。

复制构造函数 `Array` 使用成员初始化值将数组的 `size` 值复制到新数组的数据成员 `size` 中, 用 `new` 分配新数组所需的内存, 把 `new` 返回的指针赋给数据成员 `ptr`, 然后用 `assert` 测试 `new` 操作是否成功, 并递增 `arrayCount` 的值, 最后用 `for` 循环将数组的所有元素作为初始值复制到新数组中。

#### 常见编程错误 8.7

如果构造函数简单地将源对象的指针复制到目标对象的指针, 则这两个对象将指向同一块动态分配的内存块, 执行析构函数时将释放该内存块, 从而导致另外一个对象的 `ptr` 没有定义, 这种情况可能会引起严重的运行时错误。

#### 软件工程视点 8.4

通常要把构造函数、析构函数、重载的赋值运算符以及复制构造函数一起提供给使用动态内存分配的类。

第 14 行:

```
~Array(); // destructor
```

声明了类的析构函数 (第 71 行定义)。当撤消类 `Array` 的某个对象时, 自动调用析构函数。析构函数用 `delete[]` 释放在构造函数中用 `new` 动态分配的内存块, 然后递减 `arrayCount` 的值。

第 15 行:

```
int getSize() const; // return size
```

声明了读取数组大小的函数。

第 16 行:

```
const Array &operator=( const Array & ); // assign arrays
```

声明了重载的赋值运算符函数。当编译器遇到表达式:

```
integers1 = integers2;
```

就会通过产生如下代码调用函数 `operator=`:

```
integers1.operator=(integers2)
```

成员函数 `operator=` (第 82 行定义) 测试了这种赋值是否是自我赋值。如果是, 则跳过赋值操作 (即对象已经是其自身, 无需再赋值)。如果不是, 则成员函数确定两个数组长度是否相同, 如果是, 则左边 `Array` 对象的原始整数数组不重新分配。否则成员函数 `operator=` 用 `delete` 释放目标数组原先动态分配的空间, 将源数组的数据成员 `size` 复制到目标数组的 `size`, 用 `new` 分配目标数组所需的内存并将 `new` 返回的指针赋给数组的 `ptr` 成员, 用 `assert` 测试 `new` 操作是否成功, 最后再用 `for` 循环将源数组的每一个元素复制到目标数组中。不管这种操作是否是自我赋值, 成员函数都返回当前对象 (即 `*this`), 这种处理方式允许诸如 `x=y=z` 这样的连续赋值。

#### 常见编程错误 8.8

类的对象包含指向动态分配的内存的指针, 但如果没有为它提供重载的赋值运算符和复制构造函数则会造成逻辑错误。

#### 软件工程视点 8.5

把赋值运算符定义为类的 `private` 成员可以防止将一个类对象赋给另外一个类对象。

## 软件工程视点 8.6

只要重载的赋值运算符和复制构造函数为 private, 就可以防止复制类对象。

第 17 行:

```
bool operator==( const Array & ) const;    // compare equal
```

声明了重载的相等运算符。当编译器遇到 main 函数中的如下表达式时:

```
integers1 == integers2
```

编译器通过生成如下代码来调用 operator == 成员函数:

```
integers1.operator==( integers2)
```

如果数组的 size 成员不相等, 则 operator == 成员函数立即返回 false, 否则, 成员函数开始成对比较相应的元素。如果它们全都相等, 则返回 true, 一旦发现某一对元素不同则立即返回 false。

第 21 到第 22 行:

```
bool operator!=( const Array &right ) const
{ return ! ( *this == right ); }
```

声明了重载的不相等运算符 (!=)。成员函数 operator!= 根据重载的相等运算符定义。该函数定义用重载 operator== 函数确定一个 Array 是否等于另一个 Array, 然后返回结果的相反值。这样编写 operator!= 函数使程序员可以复用 operator== 函数, 减少类中需要编写的代码量。另外, operator!= 的完整函数定义在 Array 头文件中, 使编译器可以内联 operator!= 的定义, 消除额外函数调用的开销。

第 24 到第 25 行:

```
int &operator[] ( int );                // subscript operator
const int &operator[] ( int ) const;    // subscript operator
```

声明了两个重载的下标运算符 ( 分别在第 118 和 128 行定义 )。当编译器遇到 main 函数中的如下表达式时:

```
integers1[ 5 ]
```

编译器通过生成下列代码来调用重载的 operator[] 成员函数:

```
integers1.operator[] ( 5 )
```

constArray 对象使用下标运算符时, 编译器调用 operator[] 的 const 版本。operator[] 成员函数首先测试下标是否越界。如果越界, 则程序异常中止。如果没有越界, 则对 operator== 的非 const 版本返回相应的数组元素作为引用, 以便使它能用作左值 ( 如用在赋值语句的左边 )。而对 operaor[] 的 const 版本返回右值。

第 26 行:

```
static int getArrayCount();            // return count of Arrays
```

声明了 static 成员函数 getArrayCount。即使在不存在类 Array 的对象中, 该成员函数也返回静态数据成员 arrayCount 的值。

## 8.9 类型之间的转换

大多数程序能处理各种数据类型的信息。有时候所有的操作还会集中于某一种类型上,例如,整数加整数还是整数(只要结果不是太大,能用整数表示出来)。但是,常常需要将一种类型的数据转换为另外一种类型的数据,赋值、计算、给函数传值以及从函数返回值都可能会发生这种情况。对于内部类型,编译器知道如何转换类型。程序员也可以用强制类型转换运算符实现内部类型之间的强制转换。

但是怎样转换用户自定义类型呢?编译器不知道怎样实现用户自定义类型和内部类型之间的转换,程序员必须明确地指明如何转换。这种转换可以用转换构造函数实现,也就是使用单个参数的构造函数,这种函数仅仅把其他类型(包括内部类型)的对象转换为某个特定类的对象。本章稍后要用一个转换构造函数把正常的 `char*` 类型的字符串转换为类 `String` 的对象。

转换运算符(也称为强制类型转换运算符)可以把一种类的对象转换为其他类的对象或内部类型的对象。这种运算符必须是一个非 `static` 成员函数,而不能是友元函数。

函数原型:

```
A::operator char *() const;
```

声明了一个重载的强制类型转换运算符函数,它根据用户自定义类型 `A` 的对象建立一个临时的 `char*` 类型的对象。重载的强制类型转换运算符函数不能指定返回类型(返回类型是要转换后的对象类型)。如果 `s` 是某个类对象,当编译器遇到表达式 `(char*)s` 时,会产生函数调用 `s.operator char*()`,操作数 `s` 是调用成员函数 `operator char*` 的类对象 `s`。

为了把用户自定义类型的对象转换为内部类型的对象或用户自定义的其他类型的对象,我们可以定义重载的强制类型转换运算符函数。函数原型:

```
A::operator int() const;  
A::operator otherClass() const;
```

声明了两个重载的强制类型转换运算符函数,分别用来把用户自定义类型 `A` 的对象转换为一个整数和用户自定义类型 `otherClass` 的对象。

强制类型转换运算符和转换构造函数一个很好的特点就是:当需要的时候,编译器可以为建立一个临时对象而自动地调用这些函数。例如,如果用户自定义的类 `String` 的某个对象 `s` 出现在程序中需要使用 `char*` 类型的对象的位置上,例如:

```
cout << s;
```

编译器调用重载的强制类型转换运算符函数 `operator char*` 将对象转换为 `char*` 类型,并在表达式中使用转换后的 `char*` 类型的结果。`String` 类提供该转换运算符后,不需要重载流插入运算符用 `cout` 输出 `String`。

## 8.10 实例研究: String 类

作为学习重载的练习,本节要建立一个能够处理字符串的建立和操作的类(图 8.5)。`string` 类已是 C++ 标准库中的一部分,第 19 章将详细介绍 `string` 类。现在我们用运算符重载建立一个 `String` 类。





---

```

47 void setString( const char * ); // utility function
48 };
49
50 #endif
51 // Fig. 8.5: string1.cpp
52 // Member function definitions for class String
53 #include <iostream.h>
54 #include <iomanip.h>
55 #include <string.h>
56 #include <assert.h>
57 #include "string1.h"
58
59 // Conversion constructor: Convert char * to String
60 String::String( const char *s ) : length( strlen( s ) )
61 {
62     cout << "Conversion constructor: " << s << '\n';
63     setString( s ); // call utility function
64 }
65
66 // Copy constructor
67 String::String( const String &copy ) : length( copy.length )
68 {
69     cout << "Copy constructor: " << copy.sPtr << '\n';
70     setString( copy.sPtr ); // call utility function
71 }
72
73 // Destructor
74 String::~String()
75 {
76     cout << "Destructor: " << sPtr << '\n';
77     delete [] sPtr; // reclaim string
78 }
79
80 // Overloaded = operator; avoids self assignment
81 const String &String::operator=( const String &right )
82 {
83     cout << "operator= called\n";
84
85     if ( &right != this ) { // avoid self assignment
86         delete [] sPtr; // prevents memory leak
87         length = right.length; // new String length
88         setString( right.sPtr ); // call utility function
89     }
90     else
91         cout << "Attempted assignment of a String to itself\n";
92
93     return *this; // enables cascaded assignments
94 }
95
96 // Concatenate right operand to this object and
97 // store in this object.
98 const String &String::operator+=( const String &right )
99 {
100     char *tempPtr = sPtr; // hold to be able to delete
101     length += right.length; // new String length
102     sPtr = new char[ length + 1 ]; // create space

```

```
103     assert( sPtr != 0 ); // terminate if memory not allocated
104     strcpy( sPtr, tempPtr ); // left part of new String
105     strcat( sPtr, right.sPtr ); // right part of new String
106     delete[] tempPtr; // reclaim old space
107     return *this; // enables cascaded calls
108 }
109
110 // Is this String empty?
111 bool String::operator!() const { return length == 0; }
112
113 // Is this String equal to right String?
114 bool String::operator==( const String &right ) const
115     { return strcmp( sPtr, right.sPtr ) == 0; }
116
117 // Is this String less than right String?
118 bool String::operator<( const String &right ) const
119     { return strcmp( sPtr, right.sPtr ) < 0; }
120
121 // Return a reference to a character in a String as an lvalue.
122 char &String::operator[]( int subscript )
123 {
124     // First test for subscript out of range
125     assert( subscript >= 0 && subscript < length );
126
127     return sPtr[ subscript ]; // creates lvalue
128 }
129
130 // Return a reference to a character in a String as an rvalue.
131 const char &String::operator[]( int subscript ) const
132 {
133     // First test for subscript out of range
134     assert( subscript >= 0 && subscript < length );
135
136     return sPtr[ subscript ]; // creates rvalue
137 }
138
139 // Return a substring beginning at index and
140 // of length subLength as a reference to a String object.
141 String &String::operator()( int index, int subLength )
142 {
143     // ensure index is in range and substring length >= 0
144     assert( index >= 0 && index < length && subLength >= 0 );
145
146     String *subPtr = new String; // empty String
147     assert( subPtr != 0 ); // ensure new String allocated
148
149     // determine length of substring
150     if ( ( subLength == 0 ) || ( index + subLength > length ) )
151         subPtr->length = length - index + 1;
152     else
153         subPtr->length = subLength + 1;
154
155     // allocate memory for substring
156     delete subPtr->sPtr; // delete character array from object
157     subPtr->sPtr = new char[ subPtr->length ];
158     assert( subPtr->sPtr != 0 ); // ensure space allocated
```

```

159
160 // copy substring into new String
161 strncpy( subPtr->sPtr, &sPtr[ index ], subPtr->length );
162 subPtr->sPtr[ subPtr->length ] = '\0'; // terminate String
163
164 return *subPtr; // return new String
165 }
166
167 // Return string length
168 int String::getLength() const { return length; }
169
170 // Utility function to be called by constructors and
171 // assignment operator.
172 void String::setString( const char *string2 )
173 {
174     sPtr = new char[ length + 1 ]; // allocate storage
175     assert( sPtr != 0 ); // terminate if memory not allocated
176     strcpy( sPtr, string2 ); // copy literal to object
177 }
178
179 // Overloaded output operator
180 ostream &operator<<( ostream &output, const String &s )
181 {
182     output << s.sPtr;
183     return output; // enables cascading
184 }
185
186 // Overloaded input operator
187 istream &operator>>( istream &input, String &s )
188 {
189     char temp[ 100 ]; // buffer to store input
190
191     input >> setw( 100 ) >> temp;
192     s = temp; // use String class assignment operator
193     return input; // enables cascading
194 }
195 // Fig. 8.5: fig08_05.cpp
196 // Driver for class String
197 #include <iostream.h>
198 #include "stringl.h"
199
200 int main()
201 {
202     String s1( "happy" ), s2( " birthday" ), s3;
203
204     // test overloaded equality and relational operators
205     cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
206         << "\"; s3 is \"" << s3 << "\"
207         << "\nThe results of comparing s2 and s1:"
208         << "\ns2 == s1 yields "
209         << ( s2 == s1 ? "true" : "false" )
210         << "\ns2 != s1 yields "
211         << ( s2 != s1 ? "true" : "false" )
212         << "\ns2 > s1 yields "
213         << ( s2 > s1 ? "true" : "false" )
214         << "\ns2 < s1 yields "

```

```

215         << ( s2 < s1 ? "true" : "false" )
216         << "\ns2 >= s1 yields "
217         << ( s2 >= s1 ? "true" : "false" )
218         << "\ns2 <= s1 yields "
219         << ( s2 <= s1 ? "true" : "false" );
220
221 // test overloaded String empty (!) operator
222 cout << "\n\nTesting !s3:\n";
223 if ( !s3 ) {
224     cout << "s3 is empty; assigning s1 to s3;\n";
225     s3 = s1; // test overloaded assignment
226     cout << "s3 is \"" << s3 << "\"";
227 }
228
229 // test overloaded String concatenation operator
230 cout << "\n\ns1 += s2 yields s1 = ";
231 s1 += s2; // test overloaded concatenation
232 cout << s1;
233
234 // test conversion constructor
235 cout << "\n\ns1 += \" to you\" yields\n";
236 s1 += " to you"; // test conversion constructor
237 cout << "s1 = " << s1 << "\n\n";
238
239 // test overloaded function call operator () for substring
240 cout << "The substring of s1 starting at\n"
241     << "location 0 for 14 characters, s1(0, 14), is:\n"
242     << s1( 0, 14 ) << "\n\n";
243
244 // test substring "to-end-of-String" option
245 cout << "The substring of s1 starting at\n"
246     << "location 15, s1(15, 0), is: "
247     << s1( 15, 0 ) << "\n\n"; // 0 is "to end of string"
248
249 // test copy constructor
250 String *s4Ptr = new String(s1);
251 cout << "*s4Ptr = " << *s4Ptr << "\n\n";
252
253 // test assignment (=) operator with self-assignment
254 cout << "assigning *s4Ptr to *s4Ptr\n";
255 *s4Ptr = *s4Ptr; // test overloaded assignment
256 cout << "*s4Ptr = " << *s4Ptr << '\n';
257
258 // test destructor
259 delete s4Ptr;
260
261 // test using subscript operator to create lvalue
262 s1[ 0 ] = 'H';
263 s1[ 6 ] = 'B';
264 cout << "\ns1 after s1[ 0 ] = 'H' and s1[ 6 ] = 'B' is: "
265     << s1 << "\n\n";
266
267 // test subscript out of range
268 cout << "Attempt to assign 'd' to s1[ 30 ] yields:" << endl;
269 s1[ 30 ] = 'd'; // ERROR: subscript out of range
270

```

```
271     return 0;
272 }
```

**输出结果:**

```
Conversion constructor: happy
Conversion constructor: birthday
Conversion constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""
The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 > s1 yields false
s2 < s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion constructor: to you
Destructor: to you
s1 = happy birthday to you

Conversion constructor:
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Conversion constructor:
The substring of s1 starting at
location 15, s1(15,0), is: to you

copy constructor: happy birthday to you
*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator = called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[ 0] = 'H' and s1[ 6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[ 30] yields:
Assertion failed: subscript >= 0 && subscript < length,
file String1.cpp, line 76
abnormal program termination
```

图 8.5 定义基本的 String 类

我们从 String 的内部表示开始讨论。第 44 行到第 45 行:

```
int length;           // string length
char *sPtr;           // pointer to start of string
```

声明了类的 private 数据成员。String 的对象有一个 length 字段 (表示字符串中除字符串终止符以外的字符个数) 和一个指向动态分配内存 (表示字符串) 的指针 sPtr。

现在分析一下图 8.5 中定义 String 类的头文件。下面的两行代码 (第 9 行到第 10 行):

```
friend ostream &operator<<( ostream &, const String & );
friend istream &operator>>( istream &, String & );
```

把重载的流插入运算符函数 operator<< (第 180 行定义) 和流读取运算符函数 operator>> (第 187 行定义) 声明为类的友元。这两个函数的实现是显而易见的。

第 13 行:

```
String( const char * = "" ); // conversion/default ctor
```

声明了一个转换构造函数, 该构造函数 (第 60 行定义) 有一个 const char \* 类型的参数 (默认值是空字符串)。该函数实例化了 String 的一个对象, 该对象包含了与参数相同的字符串。任何只带一个参数的构造函数都可以认为是一种转换构造函数。稍后就会看到, 当使用 char \* 参数对 String 类做任何操作时, 转换构造函数是很有用的。转换构造函数把一个 char \* 字符串转换为 String 的对象 (然后该对象要赋给目标 String 对象)。使用这种转换构造函数意味着不必再为将字符串赋给 String 的对象提供重载的赋值运算符, 编译器先自动地调用该函数建立一个包含该字符串的临时 String 对象, 然后再调用重载的赋值运算符将临时 String 对象赋给另一个 String 对象。

#### 软件工程视点 8.7

当使用转换构造函数实现隐式转换时, C++ 只会使用一个隐式的构造函数调用来试图满足重载赋值运算符的需要。通过执行一系列隐式的、用户自定义的类型转换来满足重载运算符的需要是不可能的。

在做出像 String s1("happy") 这样的声明时, 调用 String 的转换构造函数。转换构造函数计算了字符串的长度并将该长度赋给 private 数据成员 length, 然后调用 private 工具函数 setString。函数 setString (第 172 行定义) 使用 new 为 private 数据成员 sPtr 分配足够的空间, 并用 assert 来测试内存分配操作是否成功。如果成功, 则用函数 strcpy 把字符串复制到对象中。

第 14 行:

```
String( const String & ); // copy constructor
```

是一个复制构造函数 (第 67 行定义), 它通过复制已存在的 String 对象来初始化一个 String 对象。必须要小心对待这种复制操作, 避免使两个 String 对象指向同一块动态分配的内存区, 默认的成员复制更容易发生这种问题。复制构造函数除了将源 String 对象的 length 成员复制到目标 String 对象外, 其余操作和转换构造函数类似。注意, 复制构造函数为目标对象的内部字符串分配了新的存储空间, 如果它只是简单地将源对象中的 sPtr 复制到目标对象的 sPtr, 则这两个对象将指向同一块动态分配的内存块。执行一个对象的析构函数将释放该内存块, 从而使另一个对象的 sPtr 没有定义, 这种情况可能会引起严重的运行时错误。

第 15 行:

```
~String(); // destructor
```

声明了类 String 的析构函数 (第 74 行定义)。该析构函数用 delete 回收构造函数中用 new 为字符串分配的动态内存。

第 16 行:

```
const String &operator=( const String & ); // assignment
```

声明了重载的赋值运算符函数 operator = (第 81 行定义)。当编译器遇到像 string1 = string2 这样的表达式时, 就会生成函数调用:

```
string1.operator=( string2 );
```

重载的赋值运算符函数 operator = 测试了这种赋值是否为自我赋值 (正如在复制构造函数中所做的那样)。如果是自我赋值运算, 由于该对象已存在, 函数就简单地返回。如果忽略自我赋值测试, 那么函数就会立即释放目标对象所占用的空间, 这样会丢失字符串。假如不是自我赋值, 那么函数就释放目标对象所占用的内存空间, 将源对象中的 length 字段复制到目标对象并调用 setString (第 172 行) 为目标对象建立新空间, 用 assert 测试 new 操作是否成功, 最后用函数 strcpy 将源对象的字符串复制到目标对象中。不管上述赋值是否为自我赋值, 函数都返回 \*this 以确保可以连续赋值。

第 17 行:

```
const String &operator+=( const String & ); // concatenation
```

声明了重载的字符串连接运算符 (第 98 行定义)。当编译器遇到 main 函数中的表达式 s1 += s2 时, 生成函数调用 s1.operator+=( s2)。函数 operator += 建立一个临时指针, 该指针用来存放当前对象的字符串指针, 直到可以撤消该字符串的内存为止, 该函数还计算了连接后的字符串长度, 用 new 为字符串分配空间, 用 assert 测试 new 操作是否成功。如果成功, 则用函数 strcpy 将原先的字符串复制到分配的空间中, 然后用函数 strcat 将源对象的字符串连接到所分配的空间中, 最后再用 delete 释放该对象原来的字符串占据的空间, 返回 \*this 作为 String & 以确保运算符 += 可以连续执行。

连接 String 类型的对象和 char \* 类型的对象不需要再重载一个连接运算符, const char \* 转换构造函数将传统的字符串转换为临时的 String 类型的对象, 然后由该对象匹配现有的重载连接运算符。C++ 为实现匹配只能在一层之内执行这样的转换。在执行内部类型和类之间的转换前, C++ 还能在内部类型之间执行编译器隐式定义的类型转换。注意, 生成临时 String 对象时, 调用转换构造函数和析构函数 (见图 8.5 中 s1 += "to you" 产生的输出)。这是隐式转换期间生成和删除临时类对象时向类客户隐藏的函数调用开销的一个例子。复制构造函数按值调用传递参数和按值返回类对象时也产生类似开销。

#### 性能提示 8.2

与先执行隐式类型转换然后再执行连接操作相比, 使重载的连接运算符 += 只有一个 const char \* 类型参数的执行效率更高。隐式类型转换需要较少的代码, 出错也较少。

第 18 行:

```
bool operator!() const; // is String empty?
```



声明了重载的取非运算符（第111行定义）。该运算符通常与字符串类一起使用，测试字符串是否为空。例如，当编译器遇到表达式!string1时，就会生成函数调用：

```
string1.operator!()
```

该函数仅仅返回 length 是否等于0的测试结果：

代码行：

```
bool operator==( const String & ) const;    // test s1 == s2
bool operator<( const String & ) const;    // test s1 < s2
```

为类String声明了重载的相等运算符（第114行定义）和关系运算符（第118行定义）。其工作原理是相似的，因此我们只以重载的运算符==为例。当编译器遇到表达式string1==string2时，就会生成如下的函数调用：

```
string1.operator==( string2 )
```

如果string1等于string2，则返回true。上述运算符都用函数strcmp比较String对象中的字符串。注意我们使用C语言标准库中的函数strcmp。许多C++程序员提倡用一些重载运算符函数实现另外一些重载运算符函数，因此!=、>、<=和>=运算符都可以用operator==和operator<实现（第23行到第36行）。例如，重载函数operator>=在头文件中的实现（第35行）如下所示：

```
bool String::operator>=( const String &right ) const
{ return ! ( *this < right ); }
```

上述operator>=定义用重载的运算符<确定一个String对象是否大于或等于另一个String对象。注意!=、>、<=和>=运算符函数都在头文件中定义。编译器将这些定义内联起来，消除多余函数调用的开销。

#### 软件工程视点 8.8

通过用前面定义的成员函数实现成员函数，程序员复用代码，从而减少要编写的代码量。

第38行到第39行：

```
char &operator[]( int );                // subscript operator
const char &operator[]( int ) const;    // subscript operator
```

声明了重载的下标运算符（在第122行和第131行定义）。一个用于const String，一个用于非const String。当编译器遇到string1[0]这样的表达式时，就会生成函数调用string1.operator[](0)（根据String是否为const类型而使用相应的operator[]版本）。函数operator[]首先用assert检查下标范围。如果下标越界，则打印一个出错信息并使程序异常中止。如果下标没有越界，则非const版本的operator[]返回一个char&类型的值，它是对String对象相应字符的引用，可用作左值，修改String对象中指定的字符。而const版本的operator[]返回String对象的相应字符，这里char&可以作为右值，读取该字符值。

#### 测试与调试提示 8.1

从String类的重载下标运算符返回char引用是危险的。例如，客户可以用这个引用在字符串中任何位置插入null终止符（'\0'）。

第40行：

```
String &operator() ( int, int );    // return a substring
```

声明了重载的函数调用运算符（第 141 行定义）。在字符串类中，为了从 String 对象中选择一个子串，经常要重载该运算符。两个整数参数指定了所选定子串的起始位置和长度。如果起始位置越界或者子串长度为负，则发出错误信息。习惯上，如果子串长度为 0，则选择的子串为从选定的开始位置一直到 String 对象的末尾。例如，假设 string1 是一个包含字符串 "AEIOU" 的 String 对象，当编译器遇到表达式 string1(2, 2) 时，生成函数调用 string1.operator()(2, 2)。执行该函数调用时，产生一个包含串 "IO" 的动态分配的新 String 对象，并返回对该对象的引用。

因为函数可能会有一个冗长而复杂的参数表，所以重载的函数调用运算符()可以有很强大的功能，从而可以完成很多有意义的操作。函数调用运算符的另外一个用途是用作数组的下标符号。例如，有的程序员不愿意用 C 的两个方括号表示二维数组（如 a[b][c]），他们更喜欢重载函数调用运算符，用 a(b, c) 表示二维数组。只有当“函数名”是类 String 的对象时才能使用该运算符。

第 41 行：

```
int getLength() const;           //return string length
```

声明了返回 String 对象长度的函数。该函数（第 168 行定义）是通过返回类 String 的 private 数据值而获得字符串的长度。

读者现在应该深入到 main 函数的代码中，研究输出结果，了解每种重载运算符的用法。

## 8.11 重载 ++ 与 --

所有四种自增和自减运算符（即前置和后置的自增及自减运算符）都可以被重载。本节介绍编译器如何识别前置和后置的自增及自减运算符。

要重载既能允许前置又能允许后置的自增运算符，每个重载的运算符函数必须有一个明确的特征以使编译器能确定要使用的 ++ 版本。重载前置 ++ 的方法与重载其他前置一元运算符一样。

例如，假设要给 Date 对象 d1 增加一天，当编译器遇到前置自增表达式：

```
++d1
```

编译器就会生成成员函数调用：

```
d1.operator++()
```

该函数的函数原型为：

```
Date &operator++();
```

如果前置自增运算符函数是一个非成员函数，则当编译器遇到表达式：

```
++d1
```

编译器就会生成函数调用：

```
operator ++(d1)
```

该函数的函数原型在类 Date 中的声明形式为：

```
friend Date &operator++(Date &);
```

由于编译器必须能区分重载的前置和后置自增运算符函数,所以重载后置自增运算符遇到了一点儿困难。C++中所采用的方法是,当编译器遇到后置自增表达式:

```
dl++
```

编译器就会生成成员函数调用:

```
dl.operator++( 0 )
```

该函数的函数原型为:

```
Date operator++( int )
```

严格说来,0是一个伪值,它使运算符函数operator++在用于后置自增操作和前置自增操作时的参数表有所区别。

如果后置自增运算符函数是一个非成员函数,则当编译器遇到表达式:

```
dl++
```

编译器就生成函数调用:

```
operator++(dl,0)
```

该函数的函数原型为:

```
friend Date operator++( Date &,int );
```

再重复一遍,编辑器使用参数0区别后置自增操作和前置自增操作所用到的operator++函数的参数表。

本节所讲述的重载前置和后置自增运算符的内容同样可以用来重载前置和后置自减运算符。下一节探讨了使用重载的前置和后置自增运算符的Date类。

## 8.12 实例研究: Date 类

图8.6声明了类Date。类Date用重载的前置和后置自增运算符将一个Date对象增加1天,必要时使年、月递增。

类Date的public接口提供了以下成员函数:一个重载的流插入运算符、一个默认的构造函数、一个setDate函数、一个重载的前置自增运算符函数、一个重载的后置自增运算符函数、一个重载的加法赋值运算符(+=)、一个检测闰年的函数和一个判断是否为每月最后一天的函数。

```
1 // Fig. 8.6: date1.h
2 // Definition of class Date
3 #ifndef DATE1_H
4 #define DATE1_H
5 #include <iostream.h>
6
7 class Date {
8     friend ostream &operator<<( ostream &, const Date & );
9
10 public:
11     Date( int m = 1, int d = 1, int y = 1900 ); // constructor
12     void setDate( int, int, int ); // set the date
13     Date &operator++(); // preincrement operator
14     Date operator++( int ); // postincrement operator
```

```

15  const Date &operator+=( int ); // add days, modify object
16  bool leapYear( int );         // is this a leap year?
17  bool endOfMonth( int );       // is this end of month?
18
19 private:
20     int month;
21     int day;
22     int year;
23
24     static const int days[];    // array of days per month
25     void helpIncrement();       // utility function
26 };
27
28 #endif
29 // Fig. 8.6: datel.cpp
30 // Member function definitions for Date class
31 #include <iostream.h>
32 #include "datel.h"
33
34 // Initialize static member at file scope;
35 // one class-wide copy.
36 const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,
37                             31, 31, 30, 31, 30, 31 };
38
39 // Date constructor
40 Date::Date( int m, int d, int y ) { setDate( m, d, y ); }
41
42 // Set the date
43 void Date::setDate( int mm, int dd, int yy )
44 {
45     month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
46     year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
47
48     // test for a leap year
49     if ( month == 2 && leapYear( year ) )
50         day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
51     else
52         day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
53 }
54
55 // Preincrement operator overloaded as a member function.
56 Date &Date::operator++()
57 {
58     helpIncrement();
59     return *this; // reference return to create an lvalue
60 }
61
62 // Postincrement operator overloaded as a member function.
63 // Note that the dummy integer parameter does not have a
64 // parameter name.
65 Date Date::operator++( int )
66 {
67     Date temp = *this;
68     helpIncrement();
69
70     // return non-incremented, saved, temporary object
71     return temp; // value return; not a reference return

```

```
72 }
73
74 // Add a specific number of days to a date
75 const Date &Date::operator+=( int additionalDays )
76 {
77     for ( int i = 0; i < additionalDays; i++ )
78         helpIncrement();
79
80     return *this;    // enables cascading
81 }
82
83 // If the year is a leap year, return true;
84 // otherwise, return false
85 bool Date::leapYear( int y )
86 {
87     if ( y % 400 == 0 || ( y % 100 != 0 && y % 4 == 0 ) )
88         return true;    // a leap year
89     else
90         return false;    // not a leap year
91 }
92
93 // Determine if the day is the end of the month
94 bool Date::endOfMonth( int d )
95 {
96     if ( month == 2 && leapYear( year ) )
97         return d == 29;    // last day of Feb. in leap year
98     else
99         return d == days[ month ];
100 }
101
102 // Function to help increment the date
103 void Date::helpIncrement()
104 {
105     if ( endOfMonth( day ) && month == 12 ) {    // end year
106         day = 1;
107         month = 1;
108         ++year;
109     }
110     else if ( endOfMonth( day ) ) {    // end month
111         day = 1;
112         ++month;
113     }
114     else    // not end of month or year; increment day
115         ++day;
116 }
117
118 // Overloaded output operator
119 ostream &operator<<( ostream &output, const Date &d )
120 {
121     static char *monthName[ 13 ] = { "", "January",
122         "February", "March", "April", "May", "June",
123         "July", "August", "September", "October",
124         "November", "December" };
125
126     output << monthName[ d.month ] << " "
127         << d.day << ", " << d.year;
128 }
```

```
129     return output;    // enables cascading
130 }
131 // Fig. 8.6: fig08_06.cpp
132 // Driver for class Date
133 #include <iostream.h>
134 #include "date1.h"
135
136 int main()
137 {
138     Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
139     cout << "d1 is " << d1
140          << "\nd2 is " << d2
141          << "\nd3 is " << d3 << "\n\n";
142
143     cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";
144
145     d3.setDate( 2, 28, 1992 );
146     cout << "  d3 is " << d3;
147     cout << "\n++d3 is " << ++d3 << "\n\n";
148
149     Date d4( 3, 18, 1969 );
150
151     cout << "Testing the preincrement operator:\n"
152          << "  d4 is " << d4 << '\n';
153     cout << "++d4 is " << ++d4 << '\n';
154     cout << "  d4 is " << d4 << "\n\n";
155
156     cout << "Testing the postincrement operator:\n"
157          << "  d4 is " << d4 << '\n';
158     cout << "d4++ is " << d4++ << '\n';
159     cout << "  d4 is " << d4 << endl;
160
161     return 0;
162 }
```

**输出结果:**

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900
```

```
d2 += 7 is January 3, 1993
```

```
  d3 is February 28, 1992
++d3 is February 29, 1992
```

```
Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969
```

```
Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```

图 8.6 重载自增运算符的 Date 类

main 函数中的驱动程序建立了几日期对象,包括:初始化为1990年1月1日的d1,初始化为1992年12月27日的d2以及初始化为一个非法日期的d3。Date的构造函数调用函数 setDate 检测月、日和年的合法性。如果月是非法的则置为1,年是非法的则置为1900,日是非法的则置为1。

驱动程序用重载的流插入运算符输出所建立的每一个Date对象。程序用重载的运算符+=将对象d2增加7天,用函数 setDate 将对象d3设置为1992年2月28日,接着将一个新对象d4设置为1969年3月18日并用重载的前置自增运算符将d4增加1天。为证实执行过程的正确性,在执行前置自增操作的前后分别输出了日期。最后,用重载的后置自增运算符将对象d4增加一天。为了证实执行的过程的正确性,在执行后置自增操作的前后也分别输出了日期。

重载前置自增运算符是简明直接的,前置自增运算符调用 private 工具函数 helpIncrement 来执行实际的自增运算。函数 helpIncrement 必须要处理日期的边界情况,因为对某月的日期加1时,它可能已经达到了最大值,此时需要将月份加1并把日期置为1,如果月份已经是12,则必须将年份加1而月份置为1。函数 helpIncrement 使用函数 leapYear 和 end of Month 正确地递增日期。

重载的前置自增运算符返回对当前对象 Date (已自增)的引用。这是因为当前对象的 \*this 作为 Date & 而返回。

重载后置自增运算符需要一点儿技巧。为模拟后置操作,函数必须返回该 Date 对象未自增的副本。在进入 operator++ 时,函数先把当前对象 (\*this) 保存在 temp 中,然后调用 helpIncrement 递增当前的 Date 对象,最后返回未递增的对象在 temp 中的副本。注意这个函数不能返回对局部 Date 对象 temp 的引用,因为声明该对象的函数退出时删除了局部变量。这样,声明这个函数的返回类型为 Date & 将返回不复存在的对象的引用。返回局部变量的引用是个常见的错误,一些编译器会发出警告。

## 小结

- 运算符<<在C++中有多种用途,既可以用作流插入运算符又可以用作左移位运算符,这是运算符重载的一个范例。同样,运算符>>也是C++中的一个重载运算符,它既可以用作流读取运算符,也可以用作右移位运算符。
- 为了使运算符在不同的上下文具有不同的含义,C++允许程序员重载大多数运算符。编译器根据运算符的使用方式产生合适的代码。
- 运算符重载提高了C++的可扩展性。
- 运算符重载是通过编写函数定义实现的。函数名是由关键字operator和其后要重载的运算符符号组成的。
- 用于类的对象的运算符必须重载,但是有两种例外情形。对于相同类的两个对象使用赋值运算符而不用重载,默认的方式是复制数据成员。地址运算符(&)也无需重载就可以用于任何类的对象,它返回对象在内存中的地址。
- C++为其内部类型提供了丰富的运算符集,重载这些运算符的目的是为用户自定义的类型提供同样简洁的表达式。
- 重载不能改变运算符的优先级和结合律。
- 重载不能改变运算符操作数的个数。重载的一元运算符仍然是一元运算符,重载的二元运算符仍然是二元运算符。C++惟一的三元运算符(?:)不能被重载。

- 不能建立新的运算符符号，只有现有的运算符才能被重载。
- 运算符重载不能改变该运算符用于内部类型的对象时的含义。
- 在重载运算符()、[]、->或者=时，运算符重载函数必须声明为类的一个成员。
- 运算符函数既可以是成员函数，也可以是非成员函数。
- 当运算符函数是一个成员函数时，最左边的操作数必须是运算符类的一个类对象（或者对该类对象的引用）。
- 如果左边的操作数必须是一个不同的类的对象，该运算符函数必须作为一个非成员函数来实现。
- 只有当二元运算符的最左边的操作数是该类的一个对象或者当一元运算符的操作数是该类的一个对象时，才会调用运算符成员函数。
- 选择非成员函数重载运算符的另外一个原因是使运算符具有可交换性。例如，给定正确的重载运算符定义，运算符左边的参数可以是其他数据成员的对象。
- 类的一元运算符可重载为一个没有参数的非static成员函数或者带有一个参数的非成员函数，参数必须是用户自定义类型的对象或者对该对象的引用。
- 二元运算符可以重载为带有一个参数的非static成员函数或者带有两个参数的非成员函数（参数之一必须是类的对象或者是对类的对象的引用）。
- 数组下标运算符不仅仅可用于数组，还可以用于从其他各种顺序容器类（如链表、字符串、字典等等）中选择元素。此外，下标不仅仅可以是整数，还可以是字符或者字符串等等。
- 复制构造函数根据同类中的其他对象初始化一个对象。不论何时需要复制对象时都会调用复制构造函数，例如在按值调用时，或是从被调用函数返回值时。在复制构造函数中，被复制的对象是通过引用传递的。
- 编译器不知道怎样实现用户自定义类型和内部类型之间的转换，程序员必须明确地指明如何进行转换。这种转换可以用转换构造函数实现（即带有单个参数的构造函数），这种函数仅仅把其他类型的对象转换为某个特定类的对象。
- 转换运算符（又称为强制类型转换运算符）可以把一种类的对象转换为其他类的对象或内部类型的对象。这种运算符必须是一个非static成员函数，而不能是友元函数。
- 转换构造函数是带有一个参数的构造函数，用来把参数转换为构造函数所在类的对象。编译器可隐式调用这种构造函数。
- 赋值运算符是最常用的重载运算符，通常用来把一个对象赋给相同类的另外一个对象。通过使用转换构造函数，赋值运算符也能够使不同类的对象之间相互赋值。
- 在不提供重载的赋值运算符时，赋值运算符的默认行为是复制类的数据成员。在有些情况下这是允许的，但是当对象中包含指向动态分配的内存区的指针时，成员复制会导致两个不同的对象指向同一块动态分配的内存区。这样，调用其中一个对象的析构函数将释放该动态分配的内存块，如果另一个对象引用该内存区，其结果是不确定的。
- 要重载既能允许前置又能允许后置的自增运算符，每个重载的运算符函数必须有一个明确的特征，以使编译器能确定要使用的++版本。重载前置++的方法与重载其他前置一元运算符一样。向后置自增运算符函数提供第二个参数（必须是int类型）达到了把前置和后置自增运算符函数区分开来的目的。实际上，用户并没有给该特定的整数参数提供值，它仅仅是让编译器区分前置和后置自增运算符函数。



## 术语

|                                               |                      |                                    |
|-----------------------------------------------|----------------------|------------------------------------|
| cast operator function                        | 强制类型转换运算符函数          | operator <=                        |
| cascaded overloaded operators                 | 连续使用重载的运算符           | operator +                         |
| class Array                                   | Array 类              | operator +=                        |
| class Date                                    | Date 类               | operator ()                        |
| class HugeInteger                             | HugeInteger 类        | operator []                        |
| class PhoneNumber                             | PhoneNumber 类        | operator char *                    |
| class String                                  | String 类             | operator int                       |
| conversion constructor                        | 转换构造函数               | operator implemented as function   |
| conversion function                           | 转换函数                 | 将运算符实现为函数                          |
| conversion operator                           | 转换运算符                | overloadable operators             |
| conversion between class types                | 类类型之间的转换             | 可重载的运算符                            |
| conversion between built-in types and classes | 类和内部类型之间的转换          | overloaded = operator              |
| copy constructor                              | 复制构造函数               | 重载的 = 运算符                          |
| dangling pointer                              | 悬挂指针                 | overloaded == operator             |
| default memberwise copy                       | 默认的成员复制              | 重载的 == 运算符                         |
| explicit type conversions (with casts)        | 显式类型转换 (使用强制类型转换运算符) | overloaded != operator             |
| friend overloaded operator function           | 重载为友元的运算符函数          | 重载的 != 运算符                         |
| function call operator                        | 函数调用运算符              | overloaded < operator              |
| implicit type conversions                     | 隐式类型转换               | 重载的 < 运算符                          |
| member function overloaded operator           | 重载为成员函数的运算符          | overloaded <= operator             |
| memory leak                                   | 内存泄漏                 | 重载的 <= 运算符                         |
| non-overloadable operators                    | 非可重载的运算符             | overloaded > operator              |
| operator keyword                              | operator 关键字         | 重载的 > 运算符                          |
| operator overloading                          | 运算符重载                | overloaded >= operator             |
| operator !                                    |                      | 重载的 >= 运算符                         |
| operator =                                    |                      | overloaded << operator             |
| operator <<                                   |                      | 重载的 << 运算符                         |
| operator >>                                   |                      | overloaded >> operator             |
| operator --                                   |                      | 重载的 >> 运算符                         |
| operator ++                                   |                      | overloaded -- operator             |
| operator ++(int)                              |                      | 重载的 -- 运算符                         |
| operator ==                                   |                      | overloaded ++ operator             |
| operator !=                                   |                      | 重载的 ++ 运算符                         |
| operator >                                    |                      | overloaded + operator              |
| operator <                                    |                      | 重载的 + 运算符                          |
| operator >=                                   |                      | overloaded += operator             |
|                                               |                      | 重载的 += 运算符                         |
|                                               |                      | overloaded assignment (=) operator |
|                                               |                      | 重载的赋值运算符                           |
|                                               |                      | overloaded [] operator             |
|                                               |                      | 重载的 [] 运算符                         |
|                                               |                      | overloading                        |
|                                               |                      | 重载                                 |
|                                               |                      | overloading a binary operator      |
|                                               |                      | 重载二元运算符                            |
|                                               |                      | overloading a unary operator       |
|                                               |                      | 重载一元运算符                            |
|                                               |                      | postfix unary operator overloading |
|                                               |                      | 后缀一元运算符重载                          |
|                                               |                      | prefix unary operator overloading  |
|                                               |                      | 前缀一元运算符重载                          |
|                                               |                      | self assignment                    |
|                                               |                      | 自我赋值                               |
|                                               |                      | single-argument constructor        |
|                                               |                      | 单个参数构造函数                           |
|                                               |                      | string concatenation               |
|                                               |                      | 字符串连接                              |
|                                               |                      | substring                          |
|                                               |                      | 子串                                 |
|                                               |                      | user-defined conversion            |
|                                               |                      | 用户自定义转换                            |
|                                               |                      | user-defined type                  |
|                                               |                      | 用户自定义类型                            |

## 自测练习

### 8.1 填空：

- a) 设  $a$  和  $b$  是两个整型变量，我们用  $a + b$  的形式求这两个变量的和；设  $c$  和  $d$  为浮点型变量，我们用  $c + d$  的形式求这两个变量的和。显然，运算符  $+$  具有不同的用途，这是 \_\_\_\_\_ 的例子。
- b) 关键字 \_\_\_\_\_ 引出了重载运算符函数的定义。
- c) 要在类的对象上使用运算符，除了运算符 \_\_\_\_\_ 和 \_\_\_\_\_ 以外，其他的必须都要被重载。
- d) 重载不能改变运算符的 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。

### 8.2 解释 C++ 中的运算符 $<<$ 和 $>>$ 的多层含义。

### 8.3 C++ 中，在什么时候可以使用名字 `operator/`?

### 8.4 (判断对错) 在 C++ 中，只能重载已有的运算符。

### 8.5 在 C++ 中，重载运算符的优先级和原先未重载的运算符的优先级相比，哪一个优先级高?

## 自测练习答案

- 8.1 a) 运算符重载。b) `operator`。c) `=`、`&`。d) 优先级、结合律和数量。
- 8.2 根据上下文，运算符  $>>$  可能是右移位运算符，也可能是流读取运算符。同样，运算符  $<<$  可能是左移位运算符，也可能是流插入运算符。
- 8.3 运算符重载时可以使用 `operator/`，它可以是提供运算符  $/$  的重载版本的函数名。
- 8.4 正确。
- 8.5 相同。

## 练习

- 8.6 尽可能多地举出 C 和 C++ 中隐式运算符重载的例子。给出需要在 C++ 中显式重载运算符的具有说服性的例子。
- 8.7 C++ 中不能被重载的运算符有 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- 8.8 字符串连接需要两个操作数，即两个要被连接的字符串。本章介绍了如何实现将第二个 `String` 对象连接到第一个 `String` 对象右边的一个重载的连接运算符，这种连接会修改第一个 `String` 对象。在有些实际应用中，需要在不修改 `String` 参数的情况下产生一个已连接的 `String` 对象，请实现允许如下操作的 `operator+`：

```
string1 = string2 + string3;
```

- 8.9 (基本运算符重载练习) 列出 C++ 的所有可重载的运算符，并对每个可重载的运算符，列出它们在用于几个不同的类时的一种或者几种可能的含义。建议试试下面的类：
  - a) 数组
  - b) 堆栈
  - c) 字符串完成后，说明哪些运算符的含义可适用于大量的类，哪些运算符重载的价值极小，哪些运算符具有歧义性。

- 8.10 现在将上一个问题所描述的过程反过来, 列出 C++ 中每个可重载的运算符, 对于每个运算符, 列出你认为它应该代表的基本操作。如果有非常好的操作, 把它们列出来。
- 8.11 (项目) C++ 是一门正在发展中的语言, 并且总是有许多新的语言开发出来。除了现有的运算符以外, 你认为还有哪些运算符可以添加到 C++ 或者添加到像 C++ 这样既支持过程化编程又支持面向对象编程的未来语言中呢? 请将你的建议寄给 ANSI C++ 委员会。
- 8.12 重载函数调用运算符()的一个较好的例子是将以下的二维数组的下标表示方法:

```
chessBoard [ row ][ column ]
```

改为常用的表示方法:

```
chessBoard ( row, column )
```

试重载函数调用运算符()支持上述表示方法。

- 8.13 生成 DoubleSubscriptedArray 类, 与图 8.4 的 Array 类的特性相似。构造时, 类应生成任意行数和列数的数组。类用 operator() 进行双下标操作。例如, 在  $3 \times 5$  的 DoubleSubscriptedArray 数组 a 中, 用户可以用 a(1, 3) 访问行 1 列 3 的元素。记住, operator() 可以接收任何参数 (关于 operator() 的例子, 见图 8.5 的 String 类)。双下标数组的基本表达方式 rows\* columns 个元素的单下标数组。函数 operator() 应通过正确的指针算法访问数组的每个元素。实际上, operator() 应有两个版本, 一个返回 int &, 使 DoubleSubscriptedArray 的元素可以用作左值, 一个返回 const int &, 使 const DoubleSubscriptedArray 的元素可以用作右值。这个类还应提供下列运算符: ==, !=, =, << (以行和列格式输出数组) 和 >> (输入整个数组内容)。
- 8.14 重载下标运算符使之返回集合中最大的元素、次最大的元素以及第三大的元素等等。
- 8.15 考虑图 8.7 中的类 Complex, 该类可以执行复数操作, 复数的格式为:

```
realPart +imaginaryPart * i
```

其中 i 是 -1 的平方根。

- 修改该类, 使之能用重载的 >> 和 << 输入和输出复数 (当然要从 Complex 类中删除打印函数)。
- 重载乘法运算符, 使之能执行两个复数的代数乘法。
- 重载运算符 == 和 !=, 使之能比较两个复数。

```
1 // Fig. 8.7: complex1.h
2 // Definition of class Complex
3 #ifndef COMPLEX1_H
4 #define COMPLEX1_H
5
6 class Complex {
7 public:
8     Complex( double = 0.0, double = 0.0 );           // constructor
9     Complex operator+( const Complex & ) const;      // addition
10    Complex operator-( const Complex & ) const;      // subtraction
11    const Complex &operator=( const Complex & );    // assignment
12    void print() const;                               // output
13 private:
14    double real;           // real part
15    double imaginary;      // imaginary part
16 };
17
```

```
18 #endif
19 // Fig. 8.7: complex1.cpp
20 // Member function definitions for class Complex
21 #include <iostream.h>
22 #include "complex1.h"
23
24 // Constructor
25 Complex::Complex( double r, double i )
26     : real( r ), imaginary( i ) { }
27
28 // Overloaded addition operator
29 Complex Complex::operator+( const Complex &operand2 ) const
30 {
31     return Complex( real + operand2.real,
32                     imaginary + operand2.imaginary );
33 }
34
35 // Overloaded subtraction operator
36 Complex Complex::operator-( const Complex &operand2 ) const
37 {
38     return Complex( real - operand2.real,
39                     imaginary - operand2.imaginary );
40 }
41
42 // Overloaded = operator
43 const Complex& Complex::operator=( const Complex &right )
44 {
45     real = right.real;
46     imaginary = right.imaginary;
47     return *this; // enables cascading
48 }
49
50 // Display a Complex object in the form: (a, b)
51 void Complex::print() const
52 { cout << '(' << real << ", " << imaginary << ')'; }
53 // Fig. 8.7: fig08_07.cpp
54 // Driver for class Complex
55 #include <iostream.h>
56 #include "complex1.h"
57
58 int main()
59 {
60     Complex x, y( 4.3, 8.2 ), z( 3.3, 1.1 );
61
62     cout << "x: ";
63     x.print();
64     cout << "\ny: ";
65     y.print();
66     cout << "\nz: ";
67     z.print();
68
69     x = y + z;
70     cout << "\n\nx = y + z:\n";
71     x.print();
72     cout << " = ";
73     y.print();
74     cout << " + ";
```

```

75     z.print();
76
77     x = y - z;
78     cout << "\n\nx = y - z:\n";
79     x.print();
80     cout << " = ";
81     y.print();
82     cout << " - ";
83     z.print();
84     cout << endl;
85
86     return 0;
87 }

```

**输出结果:**

```

x: (0,0)
y: (4.3, 8.2)
z: (3.3, 1.1)

```

```

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

```

```

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)

```

图 8.7 演示 Complex 类

8.16 32 位整数的机器所能表示的整数范围大致是 -20 亿到 +20 亿, 在这个范围内的操作一般不会出现问題。但是有很多应用程序可能要使用超出上述范围的整数, C++ 可以满足这个需求, 这需要建立一个新的数据类型。考虑图 8.8 中的类 HugeInt, 仔细研究以后, 完成下列问題:

- a) 准确描述它是如何操作的。
- b) 该类有什么限制?
- c) 重载乘法运算符 \*。
- d) 重载除法运算符 /。
- e) 重载所有的关系运算符和相等运算符。

```

1 // Fig. 8.8: hugeint1.h
2 // Definition of the HugeInt class
3 #ifndef HUGEINT1_H
4 #define HUGEINT1_H
5
6 #include <iostream.h>
7
8 class HugeInt {
9     friend ostream &operator<<( ostream &, HugeInt & );
10 public:
11     HugeInt( long = 0 );           // conversion/default constructor
12     HugeInt( const char * );       // conversion constructor
13     HugeInt operator+( HugeInt & ); // add another HugeInt
14     HugeInt operator+( int );      // add an int
15     HugeInt operator+( const char * ); // add an int in a char *
16 private:

```

```
17     short integer[30];
18 };
19
20 #endif
21 // Fig. 8.8: hugeintl.cpp
22 // Member and friend function definitions for class HugeInt
23 #include <string.h>
24 #include "hugeintl.h"
25
26 // Conversion constructor
27 HugeInt::HugeInt( long val )
28 {
29     int i;
30
31     for ( i = 0; i <= 29; i++ )
32         integer[ i ] = 0;    // initialize array to zero
33
34     for ( i = 29; val != 0 && i >= 0; i-- ) {
35         integer[ i ] = val % 10;
36         val /= 10;
37     }
38 }
39
40 HugeInt::HugeInt( const char *string )
41 {
42     int i, j;
43
44     for ( i = 0; i <= 29; i++ )
45         integer[ i ] = 0;
46
47     for ( i = 30 - strlen( string ), j = 0; i <= 29; i++, j++ )
48         integer[ i ] = string[ j ] - '0';
49 }
50
51 // Addition
52 HugeInt HugeInt::operator+( HugeInt &op2 )
53 {
54     HugeInt temp;
55     int carry = 0;
56
57     for ( int i = 29; i >= 0; i-- ) {
58         temp.integer[ i ] = integer[ i ] +
59             op2.integer[ i ] + carry;
60
61         if ( temp.integer[ i ] > 9 ) {
62             temp.integer[ i ] %= 10;
63             carry = 1;
64         }
65         else
66             carry = 0;
67     }
68
69     return temp;
70 }
71
72 // Addition
```

**輸入結果:**

```
n1 is 7654321  
n1 is 7891234  
n1 is 99999999999999999999999999999999
```





## 第9章 继 承

### 教学目标

- 能通过继承现有的类建立新类
- 了解继承是如何提高软件的可复用性
- 了解基类和派生类的概念
- 能够用多重继承从多个基类派生出新类

### 9.1 简介<sup>①</sup>

本章和下一章要讨论面向对象的程序设计的两个极其重要的功能——继承( inheritance )和多态性( polymorphism )。继承是软件复用的一种形式,实现这种方法是从现有的类建立新类。新类继承了现有类的属性和行为,并且为了使新类具有自己所需的功能,新类还要对这些属性和行为予以修饰。软件复用缩短了程序的开发时间,促使开发人员复用已经测试和调试好的高质量的软件,减少了系统投入使用后可能出现的问题。所有这些都是激动人心的。利用多态性可以编写出对现有的各种类和将要实现的类予以加工的程序。继承和多态性是处理复杂软件的一种很有效的技术。

在建立一个新类时,程序员可以让新类继承预定义基类( base class )的数据成员和成员函数,而不必重新编写新的数据成员和成员函数。这种新类称为派生类( derived class )。派生类本身也可能成为未来派生类的基类。对于单一继承,派生类只有一个基类。对于多重继承,派生类常常是从多个基类派生出来的,这些基类之间可能毫无关系。单一继承比较简单,我们介绍几个例子,使读者能很快成为专家。多重继承更复杂,也更容易出错,因此我们只显示简单的例子,建议读者在进一步深造之后再利用这种功能。

派生类通常添加了其自身的数据成员和成员函数,因而通常比基类大得多。派生类比基类更具体,它代表了一组外延较小的对象。对于单一继承,派生类和基类有相同的起源。继承的真正魅力在于能够添加基类所没有的特点以及取代和改进从基类继承来的特点。

C++ 提供三种继承: public、protected 和 private。本章以介绍 public 为主,附带介绍另外两种。第 15 章将介绍如何用 private 继承代替复合。第三种形式是 protected 继承,是 C++ 中的新生事物,用得还不多。在 public 继承中,派生类的对象也是其基类的对象,但是反过来基类对象不是其派生类的对象。本章要利用“派生类对象是基类的对象”这一关系完成一些有趣的操作。例如,把各种不同但又通过继承而相关的对象连成基类对象的链表,它允许用通常的方法处理各种对象。在本章和下一章中就会看到,这是面向对象程序设计的一种重要的功能。

本章介绍了一种新的成员访问控制形式,即 protected 访问。派生类及其友元允许访问 protected 基类成员,而其他函数则不行。

---

<sup>①</sup> 注意:本章和第 10 章介绍的许多方法已经在 C++ 体系中逐渐改变,因为 ANSI/ISO 草案标准中指定了新的方法,第 21 章将介绍这些新技术,如运行时的类型信息( RTTI )。

开发软件系统的经验表明,软件系统中的大部分代码都是处理紧密相关的特殊情况。由于设计者和程序员的精力都集中于这些特殊情况,因而很难在这种系统中看到“大手笔”的程序作品。面向对象的程序设计提供了几种“见树木而知森林”的方法,有时把这个过程称为抽象。

如果一个程序中有多种密切相关的特殊情况,通常的做法是用 switch 语句来区分各种特殊情况,然后对每种情况提供处理逻辑。第 10 章将讲述如何通过继承和多态性用更简单的逻辑取代 switch 逻辑。

我们要区别“是一个对象”和“有一个对象”的差别(以下简称“是”关系和“有”关系)。“是”关系是一种继承,在“是”关系中,派生类的对象也以作为基类的对象处理。而“有”关系是一种复合(见图 7.4),在这种关系中,一个类的对象拥有作为其成员的其他类的对象。

派生类不能访问其基类的 private 成员,否则会破坏基类的封装性。但是,派生类能够访问基类的 public 成员和 protected 成员。基类中不应该让派生类通过继承而访问的成员要在基类中声明为 private。派生类只能通过基类 public 和 protected 接口提供的访问函数访问基类的 private 成员。

继承存在的一个问题是派生类会继承它无需拥有或者不应该拥有的基类 public 成员函数。基类中不适合于派生类的成员可以在派生类中重新加以定义。有些情况下,不适合用 public 继承。

继承所最具有吸引力的特点是新类可以从现有的类库中继承。项目开发可以开发出自己的类库,也可以利用已广为使用的类库。基于这种观点,将来有一天,软件也可以像当今的硬件一样用标准的可复用组件进行构造。未来需要功能更强的软件,软件的这种开发方式正可以迎接这种挑战。

## 9.2 继承：基类和派生类

一个类的对象经常会是另一个类的对象。例如,矩形当然是四边形(正方形、平行四边形和梯形也是这样),因此可以说矩形类 Rectangle 是从四边形类 Quadrilateral 继承而来的。在本例中,类 Quadrilateral 叫做基类,类 Rectangle 称为派生类。矩形是四边形的一种特殊类型,但是要说四边形是矩形则是不正确的。图 9.1 示例了几个简单的继承例子。

| 基类       | 派生类                  |
|----------|----------------------|
| Student  | GraduateStudent      |
|          | UndergraduateStudent |
| Shape    | Circle               |
|          | Triangle             |
|          | Rectangle            |
| Loan     | CarLoan              |
|          | HomeImprovementLoan  |
|          | MortgageLoan         |
| Employee | FacultyMember        |
|          | StaffMember          |
| Account  | CheckingAccount      |
|          | SavingsAccount       |

图 9.1 几个简单的继承例子

其他的面向对象程序设计语言使用了不同的术语。例如,在继承方面,Smalltalk 语言把基类称为超类,派生类叫做子类。

因为由继承而产生的派生类通常比基类大,所以超类和子类这样的术语似乎是不合适的,本书没有使用这些术语。由于派生类对象可以看成基类的对象,因此基类有更多相关对象,而派生类的相关对象更少,因此可以把基类理解为“超类”,派生类理解为“子类”。

继承形成了树状层次结构,基类和它的派生类构成了一种层次关系。一个类可以单独存在,但是当利用继承机制使用该类时,该类就成为给其他类提供属性和行为的基类,或者成为继承其他类的属性和行为的派生类。

下面是一个简单的继承层次结构。一个典型的大学社区有成千上万个人,他们是社区的成员。这些人由雇员 (employee) 和学生 (student) 组成。雇员又分为学院成员 (faculty) 和职员 (staff), 学院成员既可能是校长和系主任等等的管理者 (administrator), 也可能是教员 (teacher)。这种关系构成的继承层次结构如图 9.2 所示。注意有些行政人员也任了课, 因此我们用多重继承构成 AdministratorTeacher 类。由于学生常常在学校打工, 职工也常常去修课, 因此还可以用多重继承构成 EmployeeStudent 类。

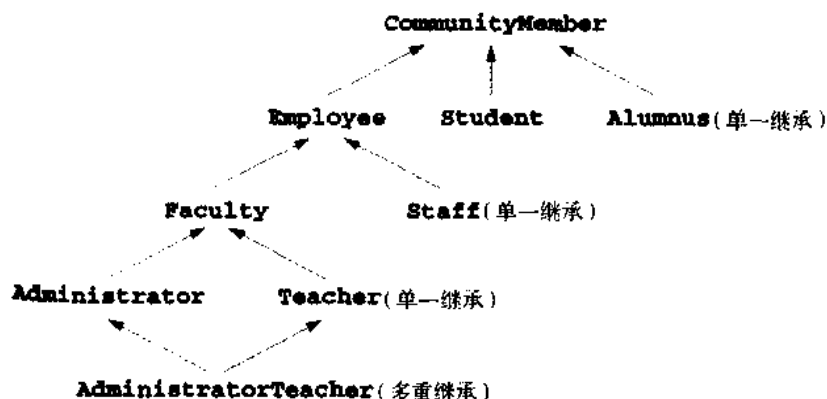


图 9.2 大学社区成员的继承层次结构

另外一个实际存在的继承层次结构是像图 9.3 那样的 Shape 层次结构。初次学习面向对象程序设计的学生都认为现实世界中存在着大量具有层次结构的实例,也正因为如此,这些学生从来没有认真思考过现实世界中的这种层次结构是如何分门别类的,所以应该在这方面好好思考一下。

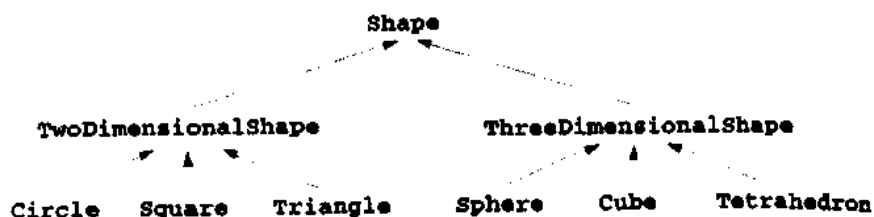


图 9.3 类 Shape 的部分层次结构

为了说明类 CommissionWorker 是从类 Employee 派生而来的,类 CommissionWorker 通常要作如下形式的定义:

```

class CommissionWorker : public Employee {
    ...
};
  
```

上述继承方法称为public继承( public inheritance ),这种类型的继承是最常用的。本章还要讨论private继承( private inheritance )和protected继承( protected inheritance )。对于 public 继承来说,基类的 public 成员和 protected 成员可以分别作为派生类的 public 成员和 protected 成员而被继承。

用类似的方法处理基类对象和派生类对象是可能的。基类的属性和行为表述了基类对象及派生类对象的共性。从基类public派生出来的所有对象都可以作为基类对象处理。我们将研究很多例子。在这些例子中,我们可以利用这种关系很容易地设计程序,而非面向对象的语言(如C语言)就做不到这一点。

## 9.3 protected 成员

基类的public成员能够被程序中所有函数访问,private成员只能被基类的成员函数和友元访问。

protected 访问是 public 访问和 private 访问之间的中间层次。基类的 protected 成员只能被基类的成员和友元以及派生类的成员和友元访问。派生类成员简单地使用成员名就可以引用基类的public成员和 protected 成员。注意 protected 数据破坏了封装,基类 protected 成员改变时,所有派生类都要修改。

### 软件工程视点 9.1

一般来说,声明private类的数据成员和使用protected方式只能是系统要满足特定性能要求时的“最后一招”。

## 9.4 把基类指针强制转换为派生类指针

公有派生类的对象可作为其相应基类的对象处理,这使得一些有意义的操作成为可能。例如,从某个特定基类派生出来的各种类,尽管这些类的对象彼此之间互不相同,但是仍然能够建立这些对象的链表,只要把这些对象作为基类对象处理就可以了。然而反过来是不行的,基类的对象不能自动成为派生类的对象。

### 常见编程错误 9.1

将基类对象作为派生类对象处理。

程序员可以用显式类型转换把基类指针强制转换为派生类指针。但是,如果要复引用该指针,那么在转换前首先应该把它指向某个派生类的对象,这一点要小心。本节采用大多数编译器中常用的方法。第 21 章将介绍符合 ANSI/ISO C++ 草案标准的最新编译器的新特性,包括运行时类型信息( RTTI )、dynamic\_cast 和 typeid。

### 常见编程错误 9.2

把指向基类对象的指针显式地强制转换为派生类指针,然后引用该对象中并不存在的派生类的成员会导致运行时的逻辑错误。

第一个例子见图 9.4。第 1 行到第 39 行是类 Point 的定义和其成员函数的定义,第 40 行到第 94 行是类 Circle 的定义和其成员函数的定义,第 95 行到第 132 行是类的驱动程序,该程序演示了如何把派生类指针赋给基类指针和如何把基类指针强制转换为派生类指针。余下的部分是程序输出。

首先看一下类 Point 的定义。Point 的 public 接口包含成员函数 setPoint、getX 和 getY。Point 的数据成员 x 和 y 指定为 protected,从而在防止了 Point 对象的用户直接访问这些数据的同时,又能够

让派生类直接访问继承来的数据成员。如果将数据成员指定为private, 那么就要用Point的public成员函数甚至派生类来访问这些数据。注意, 由于重载的流插入运算符函数是类Point的友元, 所以Point重载流插入函数能够直接引用变量x和y。因为重载的流插入运算符函数不是类Point的成员函数, 所以需要通过对对象来引用变量x和y(即p.x和p.y)。注意这个类提供内联的public成员函数getX和getY, 因此operator<<不必成为友元就可以达到良好性能。但所需public成员函数不一定在每个类的public接口中提供, 因此最好还是建立友元。

```

1 // Fig. 9.4: point.h
2 // Definition of class Point
3 #ifndef POINT_H
4 #define POINT_H
5
6 class Point {
7     friend ostream &operator<<( ostream &, const Point & );
8 public:
9     Point( int = 0, int = 0 );      // default constructor
10    void setPoint( int, int );      // set coordinates
11    int getX() const { return x; }  // get x coordinate
12    int getY() const { return y; }  // get y coordinate
13 protected:                       // accessible by derived classes
14    int x, y;                       // x and y coordinates of the Point
15 };
16
17 #endif
18 // Fig. 9.4: point.cpp
19 // Member functions for class Point
20 #include <iostream.h>
21 #include "point.h"
22
23 // Constructor for class Point
24 Point::Point( int a, int b ) { setPoint( a, b ); }
25
26 // Set x and y coordinates of Point
27 void Point::setPoint( int a, int b )
28 {
29     x = a;
30     y = b;
31 }
32
33 // Output Point (with overloaded stream insertion operator)
34 ostream &operator<<( ostream &output, const Point &p )
35 {
36     output << '[' << p.x << ", " << p.y << ' ';
37
38     return output;    // enables cascaded calls
39 }
40 // Fig. 9.4: circle.h
41 // Definition of class Circle
42 #ifndef CIRCLE_H
43 #define CIRCLE_H
44
45 #include <iostream.h>
46 #include <iomanip.h>

```

```

47 #include "point.h"
48
49 class Circle : public Point { // Circle inherits from Point
50     friend ostream &operator<<( ostream &, const Circle & );
51 public:
52     // default constructor
53     Circle( double r = 0.0, int x = 0, int y = 0 );
54
55     void setRadius( double ); // set radius
56     double getRadius() const; // return radius
57     double area() const;      // calculate area
58 protected:
59     double radius;
60 };
61
62 #endif
63 // Fig. 9.4: circle.cpp
64 // Member function definitions for class Circle
65 #include "circle.h"
66
67 // Constructor for Circle calls constructor for Point
68 // with a member initializer then initializes radius.
69 Circle::Circle( double r, int a, int b )
70     : Point( a, b ) // call base-class constructor
71 { setRadius( r ); }
72
73 // Set radius of Circle
74 void Circle::setRadius( double r )
75     { radius = ( r >= 0 ? r : 0 ); }
76
77 // Get radius of Circle
78 double Circle::getRadius() const { return radius; }
79
80 // Calculate area of Circle
81 double Circle::area() const
82     { return 3.14159 * radius * radius; }
83
84 // Output a Circle in the form:
85 // Center = [ x, y ]; Radius = #.##
86 ostream &operator<<( ostream &output, const Circle &c )
87 {
88     output << "Center = " << static_cast< Point >( c )
89         << "; Radius = "
90         << setiosflags( ios::fixed | ios::showpoint )
91         << setprecision( 2 ) << c.radius;
92
93     return output; // enables cascaded calls
94 }
95 // Fig. 9.4: fig09_04.cpp
96 // Casting base-class pointers to derived-class pointers
97 #include <iostream.h>
98 #include <iomanip.h>
99 #include "point.h"
100 #include "circle.h"
101

```

```

102 int main()
103 {
104     Point *pointPtr = 0, p( 30, 50 );
105     Circle *circlePtr = 0, c( 2.7, 120, 89 );
106
107     cout << "Point p: " << p << "\nCircle c: " << c << '\n';
108
109     // Treat a Circle as a Point (see only the base class part)
110     pointPtr = &c;    // assign address of Circle to pointPtr
111     cout << "\nCircle c (via *pointPtr): "
112         << *pointPtr << '\n';
113
114     // Treat a Circle as a Circle (with some casting)
115     pointPtr = &c;    // assign address of Circle to pointPtr
116
117     // cast base-class pointer to derived-class pointer
118     circlePtr = static_cast< Circle * >( pointPtr );
119     cout << "\nCircle c (via *circlePtr):\n" << *circlePtr
120         << "\nArea of c (via circlePtr): "
121         << circlePtr->area() << '\n';
122
123     // DANGEROUS: Treat a Point as a Circle
124     pointPtr = &p;    // assign address of Point to pointPtr
125
126     // cast base-class pointer to derived-class pointer
127     circlePtr = static_cast< Circle * >( pointPtr );
128     cout << "\nPoint p (via *circlePtr):\n" << *circlePtr
129         << "\nArea of object circlePtr points to: "
130         << circlePtr->area() << endl;
131     return 0;
132 }

```

**输出结果:**

```

Point p: [ 30, 50]
Circle c: Center = [ 120, 89]; Radius = 2.70

Circle c (via *pointPtr):[ 120,89]

Circle c(via *circlePtr):
Center = [ 120,89];Radius = 2.70
Area of c (via circlePtr): 22.90

Point p(via *circlePtr):
Center = [ 30, 50]; Radius = 0.00
Area of object circlePtr points to: 0.00

```

图 9.4 把基类指针强制转换为派生类指针

类 Circle 继承了类 Point，类定义的第一行指定了这种继承是 public 继承：

```
class Circle : public Point {    // Circle inherits from Point
```

类定义首部中的冒号(:)表示继承,关键字 public 指明了继承的类型(见 9.7 节)。类 Point 的所有 public 和 protected 成员分别作为类 Circle 的 public 和 protected 成员而被继承。这意味着 Circle 的 public 接口包括了类 Point 的 public 成员函数以及其自身的成员函数 area、setRadius 和 getRadius。

类 Circle 的构造函数必须调用类 Point 的构造函数来初始化 Circle 对象中的 Point 基类部分。这是用成员初始化值实现的(见第 7 章):

```
Circle::Circle( double r, int a, int b)
    : Point( a, b )           // call base-class constructor
```

构造函数首部的第二行函数通过类名调用类 Point 的构造函数。为初始化基类成员 x 和 y, Circle 的构造函数把 a、b 的值传递给 Point 的构造函数。如果 Circle 构造函数不显式调用 Point 构造函数,则用 x 和 y 的默认值(0 和 0)隐式调用默认的 Point 构造函数。如果这种情况下 Point 类不提供默认构造函数,则编译器产生语法错误。注意, Circle 重载 operator<< 函数可以输出 Circle 中的 Point 部分,只要将 Circle 引用 c 强制转换为 Point,从而对 Point 调用 operator<< 用正确的 Point 格式输出 x 和 y 坐标。

驱动程序先建立了一个指向 Point 对象的指针 pointPtr 并将 Point 对象实例化为 p, 然后建立一个指向 Circle 对象的指针 circlePtr 并将 Circle 对象实例化为 c。为证实初始化的正确性,程序分别用 Point 和 Circle 重载的流插入运算符输出了这两个对象的信息。然后,驱动程序将派生类指针(对象 c 的地址)赋给基类指针 pointPtr 并用 Point 的 operator<< 输出 Circle 的对象 c, 并复引用指针 \* pointPtr。注意只显示 Circle 对象 c 的 Point 部分。对 public 继承,总是可以将派生类指针赋给基类,因为派生类对象也是基类对象。基类指针只“看到”派生类对象的基类部分。编译器进行派生类指针向基类指针的隐式转换。

随后,程序将派生类指针(对象 c 的地址)赋给基类指针 pointPtr, 并将 pointPtr 强制转换回 Circle \* 类型,强制转换后的结果赋给指针 circlePtr。使用 Circle 重载流插入运算符输出 Circle 的对象 c 并复引用指针 \* circlePtr。然后通过指针 circlePtr 输出 Circle 对象 c 的面积。因为该指针一直指向 Circle 对象,所以输出了该对象的合法面积值。

因为把基类指针直接赋给派生类指针蕴含着危险性,所以编译器不允许这么做,也不执行隐式转换。使用显式类型转换是告诉编译器程序员已经知道了这种危险性。正确地使用指针是程序员的责任,因此编译器允许有危险的转换。

接着,程序演示了将基类指针(对象 p 的地址)赋给基类指针 pointPtr, 并将 pointPtr 强制转换为 Circle \* 类型,强制转换操作的结果赋给了 circlePtr。Point 对象 p 用 Circle 的 operator<< 输出,并复引用指针 \* circlePtr。注意半径元素输出为 0(实际上不存在,因为 circlePtr 实际上针对 Point 对象)。将 Point 作为 Circle 输出就会导致 radius 为未定义的值(这里刚好为 0),因为指针总是指向 Point 对象。Point 对象没有 radius 成员,因此输出 circlePtr 所指 radius 数据成员内存地址中的值。circlePtr 所指对象的面积(Point 对象 p)也是通过 circlePtr 输出。注意面积值为 0.00,这是根据 radius“未定义”的值算出的。显然,访问不存在的数据成员是很危险的。调用不存在的成员函数可能使程序崩溃。

本节介绍指针转换的机制。为下一章介绍多态与面向对象编程打下了基础。

## 9.5 使用成员函数

当从基类派生出一个派生类时,派生类的成员函数可能需要访问基类的某些成员函数。



### 软件工程视点 9.2

派生类不能直接访问其基类的 private 成员。

这是 C++ 中关键的软件工程视点。如果派生类能访问其基类的 private 成员，那么就会破坏基类的封装性。隐藏 private 成员有助于测试、调试和正确地修改系统。如果派生类能访问其基类的 private 成员，那么从派生类派生出的类也应该能访问这些成员，这样就会传递对 private 数据的访问权，从而使封装所带来的益处在整个类层次上损失殆尽。

## 9.6 在派生类中重定义基类成员

派生类可以通过提供同样签名的新版本（如果签名不同，则是函数重载而不是函数重定义）重新定义基类成员函数。派生类引用该函数时会自动选择派生类中的版本。作用域运算符可用来从派生类中访问基类的该成员函数的版本。

### 常见编程错误 9.3

派生类中重新定义基类的成员函数时，为完成某些附加工作，派生类版本通常要调用基类中的该函数版本。不使用作用域运算符会由于派生类成员函数实际上调用了自身而引起无穷递归。这样会使系统用光内存，是致命的运行时错误。

考察一个简单的类 Employee，它存储雇员的姓（成员 firstName）和名（成员 lastName）。这种信息对于所有雇员（包括 Employee 的派生类的雇员）是很普遍的。现在假设从雇员类 Employee 派生出了小时工类 HourlyWorker、计件工类 PieceWorker、老板类 Boss 和销售员类 CommissionWorker。小时工每周工作 40 小时，超过 40 小时部分的报酬是平时的 1.5 倍；计件工是按生产的工作计算报酬的，每件的报酬是固定的，假设他只生成一种类型的工件，因而类 PieceWorker 的 private 数据成员是生产的工件数量和每件的报酬；老板每周有固定的薪水；销售员每周有小部分固定的基本工资加上其每周销售额的固定百分比。为简单起见，此处只研究类 Employee 和派生类 HourlyWorker。

本章的第二个例子见图 9.5。第 1 行到 47 行分别是类 Employee 的定义和其成员函数的定义，第 48 行到 94 行分别是类 HourlyWorker 的定义和其成员函数的定义，第 95 行到结束是类继承层次 Employee/HourlyWorker 的驱动程序，该程序很简单，仅仅建立并初始化了类 HourlyWorker 的对象，然后调用类 HourlyWorker 的成员函数 print 输出对象的数据。

```
1 // Fig. 9.5: employ.h
2 // Definition of class Employee
3 #ifndef EMPLOY_H
4 #define EMPLOY_H
5
6 class Employee {
7 public:
8     Employee( const char *, const char * ); // constructor
9     void print() const; // output first and last name
10    ~Employee(); // destructor
11 private:
12    char *firstName; // dynamically allocated string
13    char *lastName; // dynamically allocated string
14 };
15
```

```
16 #endif
17 // Fig. 9.5: employ.cpp
18 // Member function definitions for class Employee
19 #include <string.h>
20 #include <iostream.h>
21 #include <assert.h>
22 #include "employ.h"
23
24 // Constructor dynamically allocates space for the
25 // first and last name and uses strcpy to copy
26 // the first and last names into the object.
27 Employee::Employee( const char *first, const char *last )
28 {
29     firstName = new char[ strlen( first ) + 1 ];
30     assert( firstName != 0 ); // terminate if not allocated
31     strcpy( firstName, first );
32
33     lastName = new char[ strlen( last ) + 1 ];
34     assert( lastName != 0 ); // terminate if not allocated
35     strcpy( lastName, last );
36 }
37
38 // Output employee name
39 void Employee::print() const
40 { cout << firstName << ' ' << lastName; }
41
42 // Destructor deallocates dynamically allocated memory
43 Employee::~Employee()
44 {
45     delete [] firstName; // reclaim dynamic memory
46     delete [] lastName;  // reclaim dynamic memory
47 }
48 // Fig. 9.5: hourly.h
49 // Definition of class HourlyWorker
50 #ifndef HOURLY_H
51 #define HOURLY_H
52
53 #include "employ.h"
54
55 class HourlyWorker : public Employee {
56 public:
57     HourlyWorker( const char*, const char*, double, double );
58     double getPay() const; // calculate and return salary
59     void print() const;    // overridden base-class print
60 private:
61     double wage;           // wage per hour
62     double hours;         // hours worked for week
63 };
64
65 #endif
66 // Fig. 9.5: hourly.cpp
67 // Member function definitions for class HourlyWorker
68 #include <iostream.h>
69 #include <iomanip.h>
70 #include "hourly.h"
```

```

71
72 // Constructor for class HourlyWorker
73 HourlyWorker::HourlyWorker( const char *first,
74                             const char *last,
75                             double initHours, double initWage )
76     : Employee( first, last )    // call base-class constructor
77 {
78     hours = initHours; // should validate
79     wage = initWage;   // should validate
80 }
81
82 // Get the HourlyWorker's pay
83 double HourlyWorker::getPay() const { return wage * hours; }
84
85 // Print the HourlyWorker's name and pay
86 void HourlyWorker::print() const
87 {
88     cout << "HourlyWorker::print() is executing\n\n";
89     Employee::print(); // call base-class print function
90
91     cout << " is an hourly worker with pay of $"
92          << setiosflags( ios::fixed | ios::showpoint )
93          << setprecision( 2 ) << getPay() << endl;
94 }
95 // Fig. 9.5: fig.09_05.cpp
96 // Overriding a base-class member function in a
97 // derived class.
98 #include <iostream.h>
99 #include "hourly.h"
100
101 int main()
102 {
103     HourlyWorker h( "Bob", "Smith", 40.0, 10.00 );
104     h.print();
105     return 0;
106 }

```

**输出结果:**

```

HourlyWorker::print() is executing
Bob Smith is an hourly worker with pay of $400.00

```

图 9.5 在派生类中重新定义基类的成员函数

类 `Employee` 的定义由两个 `private char *` 类型的数据成员 (`firstName` 和 `lastName`) 和三个成员函数 (构造函数、析构函数和 `print` 函数) 组成。构造函数接收两个字符串, 并动态分配存储字符串的字符数组。宏 `assert` (见第 18 章) 用来确定是否为 `firstName` 和 `lastName` 分配了内存。如果没有, 程序终止并返回一条出错信息, 该信息指出了被测试的条件以及条件所在的行号和文件。由于 `Employee` 的数据是 `private` 类型, 所以只能用成员函数 `print` 访问数据, 函数 `print` 非常简单, 仅仅输出雇员的姓和名。析构函数将动态分配的内存交还给系统 (防止内存泄漏)。

类 `HourlyWorker` 对类 `Employee` 的继承是 `public` 继承。类定义的第一行指定了这种继承方式:

```
class HourlyWorker : public Employee
```

HourlyWorker的public接口包括Employee的函数print和HourlyWorker的成员函数getPay和print。注意,类HourlyWorker定义了其自身的print函数(使用同样的函数原型Employee: print()),所以类HourlyWorker有权访问两个print函数。类HourlyWorker还包含用来计算雇员的每周薪水的private数据成员wage和hours。

HourlyWorker的构造函数用成员初始化值语法将字符串first和last传递给Employee的构造函数,从而初始化了基类的成员,然后再初始化成员wage和hours。成员函数getPay用来计算HourlyWorker的工资。

类HourlyWorker的成员函数print重新定义Employee的print成员函数。为提供更多的功能而在派生类中重新定义基类的成员函数是常有的事。被重新定义的函数有时候为执行一些新任务而要调用基类中的函数版本。在本例中,派生类函数print调用基类Employee的print函数输出了雇员的名字(基类print函数是惟一能访问该类private数据的函数),派生类的print函数输出了雇员的工资。调用基类print函数的方法如下:

```
Employee::print();
```

因为基类函数和派生类函数的名字相同,所以必须在基类函数前使用基类名和作用域运算符,否则将调用派生类的函数版本(即类HourlyWorker的print函数调用其自身),从而导致无穷递归。

## 9.7 public、protected和private继承

从一个基类派生一个类时,继承基类的方式有三种:public、protected和private。protected继承和private继承不常用,而且使用时必须相当小心。本书中的范例都是使用public继承(第15章将介绍用private继承作为复合的另一种形式)。图9.6总结了每种继承中派生类对基类成员的访问性。第一列包含基类成员的访问说明符。

| 基类成员的<br>访问说明符 | 继承类型                                                                          |                                                                               |                                                                               |
|----------------|-------------------------------------------------------------------------------|-------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
|                | public 继承                                                                     | protected 继承                                                                  | private 继承                                                                    |
| public         | 在派生类中为 public<br><br>可以直接由任何非 static<br>成员函数、友元函数和<br>非成员函数访问                 | 在派生类中为 protected<br><br>可以直接由任何非 static<br>成员函数、友元函数访问                        | 在派生类中为 private<br><br>可以直接由任何非 static<br>成员函数、友元函数访问                          |
| protected      | 在派生类中为 protected<br><br>可以直接由任何非 static<br>成员函数、友元函数访问                        | 在派生类中为 protected<br><br>可以直接由任何非 static<br>成员函数、友元函数访问                        | 在派生类中为 private<br><br>可以直接由任何非 static<br>成员函数、友元函数访问                          |
| private        | 在派生类中隐藏<br><br>可以通过基类的 public<br>或 protected 成员函数由非<br>static 成员函数和友元函<br>数访问 | 在派生类中隐藏<br><br>可以通过基类的 public<br>或 protected 成员函数由非<br>static 成员函数和友元函<br>数访问 | 在派生类中隐藏<br><br>可以通过基类的 public<br>或 protected 成员函数由非<br>static 成员函数和友元函<br>数访问 |

图 9.6 派生类对基类成员的访问性

从 public 基类派生某个类时,基类的 public 成员会成为派生类的 public 成员,基类的 protected 成员成为派生类的 protected 成员。派生类永远也不能直接访问基类的 private 成员,但可通过基类 public 或 protected 成员间接访问。

从 protected 基类派生一个类时,基类的 public 成员和 protected 成员成为派生类的 protected 成员。从 private 基类派生一个类时,基类的 public 成员和 protected 成员成为派生类的 private 成员(例如,函数成为工具函数),private 和 protected 继承不是“是”的关系。

## 9.8 直接基类和间接基类

基类既可能是派生类的直接基类,也可能是派生类的间接基类。在声明派生类时,派生类的首部要显式地列出直接基类。间接基类不是显式地列在派生类的首部,而是沿着类的多个层次向上继承。

## 9.9 在派生类中使用构造函数和析构函数

由于派生类继承了其基类的成员,所以在建立派生类的实例对象时,必须调用基类的构造函数来初始化派生类对象的基类成员。派生类的构造函数既可以隐式调用基类的构造函数,也可以在派生类的构造函数中通过给基类提供初始化值(利用了前面所讲过的成员初始化值语法)显式地调用基类的构造函数。

派生类不继承基类的构造函数和赋值运算符,但是派生类的构造函数和赋值运算符能调用基类的构造函数和赋值运算符。

派生类的构造函数总是先调用其基类构造函数来初始化派生类中的基类成员。如果省略了派生类的构造函数,那么就由派生类的默认构造函数调用基类的默认构造函数。析构函数的调用顺序和调用构造函数的顺序相反,因此派生类的析构函数在基类析构函数之前调用。

### 软件工程视点 9.3

假设生成派生类对象,基类和派生类都包含其他类的对象,则在建立派生类的对象时,首先执行基类成员对象的构造函数,接着执行基类的构造函数,然后执行派生类的成员对象的构造函数,最后才执行派生类的构造函数。析构函数的调用次序与调用构造函数的次序相反。

### 软件工程视点 9.4

建立成员对象的顺序是对象在类定义中的声明顺序。成员初始化值的顺序不影响建立对象的顺序。

### 软件工程视点 9.5

对继承关系而言,基类构造函数的调用顺序是派生类定义中指定的继承顺序,派生类成员初始化值列表中指定的基类构造函数的顺序不影响对象的建立顺序。

图 9.7 中的程序演示了基类和派生类的构造函数及析构函数的调用顺序。程序的第 1 行到第 35 行是一个简单的 Point 类,包含一个构造函数、一个析构函数以及 protected 数据成员 x 和 y。构造函数和析构函数打印了调用它们的 Point 对象的信息。

第 36 行到第 72 行是一个简单的 Circle 类,它是通过 public 继承从 Point 类派生出来的。类 Circle 包含一个构造函数、一个析构函数以及 private 数据成员 radius。构造函数和析构函数打印了调用它们的 Circle 对象的信息。为初始化基类的数据成员 x 和 y, Circle 的构造函数用成员初始化值语法和传递变量 a 和 b 的值调用 Point 类的构造函数。

第73行到最后是层次结构 Point/Circle 的驱动程序。程序首先在 main 函数内实例化了一个 Point 对象。由于该对象在进入其范围后又立即退出其范围, 所以调用了 Point 的构造函数和析构函数。然后, 程序实例化了类 Circle 的对象 circle1。这个过程调用了类 Point 的构造函数, 从而输出了类 Circle 的构造函数传递给它的值, 随后再输出 Circle 构造函数所指定的输出内容。接着, 程序实例化了类 Circle 的对象 circle2。这个过程同样需要调用类 Point 和 Circle 的构造函数。注意, 类 Point 的构造函数在执行 Circle 构造函数之前执行。main 函数结束时, 程序为对象 circle1 和 circle2 调用析构函数。因为调用析构函数的顺序和调用构造函数的顺序相反, 所以先为对象 circle2 调用析构函数, 调用顺序是调用完类 Circle 的析构函数后, 再调用类 Point 的析构函数。为对象 circle2 调用完析构函数后, 再以相同的顺序为对象 circle1 调用析构函数。

```

1 // Fig. 9.7: point2.h
2 // Definition of class Point
3 #ifndef POINT2_H
4 #define POINT2_H
5
6 class Point {
7 public:
8     Point( int = 0, int = 0 ); // default constructor
9     ~Point(); // destructor
10 protected: // accessible by derived classes
11     int x, y; // x and y coordinates of Point
12 };
13
14 #endif
15 // Fig. 9.7: point2.cpp
16 // Member function definitions for class Point
17 #include <iostream.h>
18 #include "point2.h"
19
20 // Constructor for class Point
21 Point::Point( int a, int b )
22 {
23     x = a;
24     y = b;
25
26     cout << "Point constructor: "
27          << '[' << x << ", " << y << ']' << endl;
28 }
29
30 // Destructor for class Point
31 Point::~~Point()
32 {
33     cout << "Point destructor: "
34          << '[' << x << ", " << y << ']' << endl;
35 }
36 // Fig. 9.7: circle2.h
37 // Definition of class Circle
38 #ifndef CIRCLE2_H
39 #define CIRCLE2_H
40
41 #include "point2.h"
42

```

```
43 class Circle : public Point {
44 public:
45     // default constructor
46     Circle( double r = 0.0, int x = 0, int y = 0 );
47
48     ~Circle();
49 private:
50     double radius;
51 };
52
53 #endif
54 // Fig. 9.7: circle2.cpp
55 // Member function definitions for class Circle
56 #include "circle2.h"
57
58 // Constructor for Circle calls constructor for Point
59 Circle::Circle( double r, int a, int b )
60     : Point( a, b )    // call base-class constructor
61 {
62     radius = r; // should validate
63     cout << "Circle constructor: radius is "
64          << radius << " [ " << x << ", " << y << ']' << endl;
65 }
66
67 // Destructor for class Circle
68 Circle::~~Circle()
69 {
70     cout << "Circle destructor: radius is "
71          << radius << " [ " << x << ", " << y << ']' << endl;
72 }
73 // Fig. 9.7: fig09_07.cpp
74 // Demonstrate when base-class and derived-class
75 // constructors and destructors are called.
76 #include <iostream.h>
77 #include "point2.h"
78 #include "circle2.h"
79
80 int main()
81 {
82     // Show constructor and destructor calls for Point
83     {
84         Point p( 11, 22 );
85     }
86
87     cout << endl;
88     Circle circle1( 4.5, 72, 29 );
89     cout << endl;
90     Circle circle2( 10, 5, 5 );
91     cout << endl;
92     return 0;
93 }
```

**输出结果:**

```
Point constructor: [ 11, 22]
Point destructor: [ 11, 22]
```

```
Point  constructor: [ 72, 29]
Circle constructor: radius is 4.5 [ 72, 29]

Point  constructor: [ 5, 5]
Circle constructor: radius is 10 [ 5, 5]

Circle destructor: radius is 10 [ 5, 5]
Point  destructor: [ 5, 5]
Circle destructor: radius is 4.5 [ 72, 29]
Point  destructor: [ 72, 29]
```

图 9.7 基类和派生类的构造函数和析构函数的调用顺序

## 9.10 将派生类对象隐式转换为基类对象

尽管派生类对象也是基类对象,但是派生类类型和基类类型是不同的。在 public 继承中,派生类对象能作为基类对象处理。由于派生类具有对应每个基类成员的成员(派生类的成员通常比基类的成员多),所以把派生类的对象赋给基类对象是合理的。但是,反过来赋值会使派生类中基类不具有的成员没有定义,所以这是不允许的。尽管如此,提供正确的重载赋值运算符和(或)转换构造函数可以允许这种操作(见第 8 章)。

### 常见编程错误 9.4

把派生类对象赋给其基类对象,然后试图在新的基类对象中引用只在派生类中才有的成员是个语法错误。

注意,在本节后面提到指针时,也适用于引用。

在 public 继承中,因为派生类对象也是基类对象,所以指向派生类对象的指针可以隐式地转换为指向基类对象的指针。

基类指针和派生类指针与基类对象和派生类对象的混合和匹配有如下四种可能的方式:

1. 直接用基类指针引用基类的对象。
2. 直接用派生类指针引用派生类的对象。
3. 用基类指针引用一个派生类的对象。由于派生类的对象也是基类的对象,所以这种引用方式是安全的,但是用这种方法只能引用基类成员。如果试图通过基类指针引用那些只在派生类中才有的成员,编译器会报告语法错误。
4. 用派生类指针引用基类的对象。这种引用方式会导致语法错误。派生类指针必须先强制转换为基类指针。

### 常见编程错误 9.5

将基类指针强制转换为派生类指针,如果用该指针引用基类对象,而基类对象中没有所要引用的派生类的成员,那么这时就会发生错误。

将派生类对象作为基类对象可能是很方便的,但使用基类指针操作这些对象容易出问题。例如,在某个计算工资单的系统,我们希望能够遍历关于雇员的清单并计算出每人每周的工资。但是,使用基类指针使得程序只能调用基类的工资单计算例程(如果基类中确实存在该例程)。我们需要一种方法为每一个对象(不管它是派生类对象还是基类对象)调用正确的工资单计算例程,并且这种方法只需简单地使用基类指针。这个问题的答案是使用第 10 章介绍的虚函数和多态性。



## 9.11 关于继承的软件工程

我们可以用继承来定制现有的软件。为了把现有类定制成满足我们的需要的类,首先要继承现有类的属性和行为,然后添加和去除一些属性和行为。在C++中,派生类不必访问基类的源代码,但是需要能够连接到基类的目标代码。这种强大的功能对独立软件供应商(ISV)很有吸引力。ISV开发出具有目标代码格式的类后,他们就拥有了这些类的所有权,因而可以销售和发放使用许可证。用户拥有这些类后,在不访问源代码(所有权属于ISV)的情况下,他们就能够从这些类库中派生出新的类,所有的ISV需要为目标代码提供头文件。

### 软件工程视点 9.6

理论上,用户不需要看到所继承类的源代码。但实际上,根据发放许可证的经验,客户通常会需要源代码。程序员似乎还是不太愿意放心地把别人编写的代码放进自己的程序中。

### 性能提示 9.1

如果性能是主要考虑,则程序员可能要浏览所继承类的源代码,以便根据性能要求调整代码。

学生们很难认识到大型软件项目的设计者和实现者所面临的问题。有过开发这种项目经验的人都知道缩短软件开发过程的关键是鼓励软件复用。面向对象的程序设计普遍鼓励软件复用,而C++尤其提倡软件复用。

正是继承了实用的类库才发挥出了软件复用的最大优势。随着人们对C++的兴趣不断增长,对类库感兴趣的人也将增加。正如个人电脑的出现带动了ISV生产的套装软件日益增长,C++也必将带动类库的建立和销售。因为应用程序设计者会用这些类库建立他们自己的应用程序,所以类库设计者也将因此而获得丰厚的报偿。当前随C++编译器分发的类库倾向于一定的通用性并限制使用范围。在世界范围内开发应用于各种领域的类库的时代正在来临。

### 软件工程视点 9.7

建立一个派生类不会影响其基类的源代码和目标代码,继承这一机制保护了基类的完整性。

基类描述了共性。所有从基类派生出来的类都继承了基类的功能。在面向对象的设计过程中,设计者先寻求并提取出构成所需基类的共性,然后再通过继承从基类派生出超出基类功能的定制派生类。

### 软件工程视点 9.8

在面向对象的系统中,类常常是紧密相关的。提取出共同的属性和行为并把它放在一个基类中,然后再通过继承生成派生类。

正如非面向对象系统的设计者力图避免不必要的函数一样,面向对象系统的设计者也应该避免不必要的类。多余的类不仅会带来类管理上的问题,而且会阻碍软件的复用。理由很简单,因为用户难以在巨大的类集合中定位某个类。权衡的结果还是建立较少的类,每个类都实际增加一些功能。这样的类对于某些用户来说可能功能太丰富了一点,但是他们可以屏蔽掉多余的功能,然后使之满足自己的需要。

### 性能提示 9.2

大于功能需求的派生类可能会浪费内存和处理资源。因此应继承最接近要求的类。

注意, 因为派生类中没有列出继承来的成员, 所以浏览一组派生类的声明会令人迷惑, 但是派生类中确实存在继承来的成员。

#### 软件工程视点 9.9

派生类除了包含其基类的属性和行为外, 还能够包含附加的属性和行为。继承机制能够使基类独立于派生类编译。为了把基类与派生类中增加的属性和行为组合成派生类, 编译器只需要编译派生类中增加的属性和行为。

#### 软件工程视点 9.10

只要基类的 `public` 接口不变, 对基类的修改无需修改派生类, 但是派生类需要重新编译。

## 9.12 复合与继承的比较

我们讨论了 `public` 继承所支持的“是”关系, 还讨论把对象作为成员的“有”关系, 并举了几个例子。“有”关系通过复合现有的类建立了新类。例如, 假设有雇员类 `Employee`、生日类 `BirthDate` 和电话号码类 `TelephoneNumber`, 说雇员 (`Employee`) 是一个生日 (`BirthDate`) 或电话号码 (`TelephoneNumber`) 是不对的, 但是说雇员有生日和电话号码当然是合适的。

#### 软件工程视点 9.11

只要成员类的 `public` 接口不变, 对成员类的修改无需修改复合类, 但是复合类需要重新编译。

## 9.13 对象的“使用”关系和“知道”关系

继承和复合都提倡建立与现有的类有许多共性的新类来实现软件复用。还有其他一些方法可以利用类所提供的服务。尽管“人”不是一辆汽车, “人”也不能包含汽车, 但“人”当然可以“使用”汽车。一个函数可以简单地向对象发出函数调用来“使用”这个对象。

一个对象可以“知道”另外一个对象, 知识网中常常存在这种关系。一个对象可以包含指向对象的指针或对该对象的引用, 从而“知道”那个对象的存在。在这种情况下, 可以说一个对象和另一个对象具有“知道”关系。

## 9.14 实例研究: 类 `Point`、`Circle` 和 `Cylinder`

下面考察本章的一个练习, 即点、圆、圆柱体的层次结构。我们首先开发并使用类 `Point` (图 9.8), 然后从类 `Point` 派生出类 `Circle` (图 9.9), 最后从类 `Circle` 派生出类 `Cylinder` (图 9.10)。

图 9.8 列出了类 `Point`。图中的第 1 行到第 17 行是类 `Point` 的定义。可以看到, 类 `Point` 的数据成员为 `protected`。因此, 当从类 `Point` 派生出类 `Circle` 时, 类 `Circle` 的成员函数不必使用访问函数就能够直接引用坐标 `x` 和 `y`, 这样可使性能更好。

第 18 行到第 39 行定义了类 `Point` 的成员函数。第 40 行到第 57 行是类 `Point` 的驱动程序。程序中的 `main` 函数必须使用访问函数 `getX` 和 `getY` 读取 `protected` 数据成员 `x` 和 `y` 的值。要记住, `protected` 数据成员只能被类和其派生类的成员和友元访问。

```
1 // Fig. 9.8: point2.h
2 // Definition of class Point
3 #ifndef POINT2_H
4 #define POINT2_H
5
6 class Point {
7     friend ostream &operator<<( ostream &, const Point & );
8 public:
9     Point( int = 0, int = 0 );      // default constructor
10    void setPoint( int, int );      // set coordinates
11    int getX() const { return x; }  // get x coordinate
12    int getY() const { return y; }  // get y coordinate
13 protected:                       // accessible to derived classes
14    int x, y;                       // coordinates of the point
15 };
16
17 #endif
18 // Fig. 9.8: point2.cpp
19 // Member functions for class Point
20 #include <iostream.h>
21 #include "point2.h"
22
23 // Constructor for class Point
24 Point::Point( int a, int b ) { setPoint( a, b ); }
25
26 // Set the x and y coordinates
27 void Point::setPoint( int a, int b )
28 {
29     x = a;
30     y = b;
31 }
32
33 // Output the Point
34 ostream &operator<<( ostream &output, const Point &p )
35 {
36     output << '[' << p.x << ", " << p.y << ']' ;
37
38     return output;          // enables cascading
39 }
40 // Fig. 9.8: fig09_08.cpp
41 // Driver for class Point
42 #include <iostream.h>
43 #include "point2.h"
44
45 int main()
46 {
47     Point p( 72, 115 );      // instantiate Point object p
48
49     // protected data of Point inaccessible to main
50     cout << "X coordinate is " << p.getX()
51          << "\nY coordinate is " << p.getY();
52
53     p.setPoint( 10, 10 );
54     cout << "\n\nThe new location of p is " << p << endl;
55 }
```

```
56     return 0;
57 }
```

**输出结果:**

```
X coordinate is 72
Y coordinate is 115
```

```
The new location of p is [ 10,10 ]
```

图9.8 演示 Point 类

第二个例子是图9.9。该例子复用了图9.8中的类Point的定义和成员函数定义，分别是类Circle的定义、类Circle的成员函数的定义和类的驱动程序。类Circle对类Point的继承是public继承，这意味着类Circle的public接口包括类Point的成员函数以及Circle成员函数setRadius、getRadius和area。

注意Circle重载operator<<函数，Circle类的友元可以通过将Circle引用c强制转换为Point而输出Circle的Point部分。因此调用Point的operator<<并用相应的Point格式输出x和y坐标。

驱动程序先实例化了类Circle的一个对象，然后用“get”函数读取该对象的信息。main函数既不是类Circle的成员函数也不是其友元，因此它不能直接引用该类的protected数据。为此，程序中用“set”函数setRadius和setPoint重新设置圆的半径和圆心坐标。最后，驱动程序先将Point&（对Point对象的引用）类型的引用变量pRef初始化为Circle的对象c，然后打印出pRef，尽管它已经被初始化为一个Circle对象，但是它还“认为”自己是一个Point对象，所以该Circle对象实际上作为Point对象打印的。

```
1 // Fig. 9.9: circle2.h
2 // Definition of class Circle
3 #ifndef CIRCLE2_H
4 #define CIRCLE2_H
5
6 #include "point2.h"
7
8 class Circle : public Point {
9     friend ostream &operator<<( ostream &, const Circle & );
10 public:
11     // default constructor
12     Circle( double r = 0.0, int x = 0, int y = 0 );
13     void setRadius( double );    // set radius
14     double getRadius() const;    // return radius
15     double area() const;        // calculate area
16 protected:                    // accessible to derived classes
17     double radius;              // radius of the Circle
18 };
19
20 #endif
21 // Fig. 9.9: circle2.cpp
22 // Member function definitions for class Circle
23 #include <iostream.h>
24 #include <iomanip.h>
25 #include "circle2.h"
26
27 // Constructor for Circle calls constructor for Point
28 // with a member initializer and initializes radius
```

```

29 Circle::Circle( double r, int a, int b )
30     : Point( a, b )           // call base-class constructor
31 { setRadius( r ); }
32
33 // Set radius
34 void Circle::setRadius( double r )
35     { radius = ( r >= 0 ? r : 0 ); }
36
37 // Get radius
38 double Circle::getRadius() const { return radius; }
39
40 // Calculate area of Circle
41 double Circle::area() const
42     { return 3.14159 * radius * radius; }
43
44 // Output a circle in the form:
45 // Center = [ x, y]; Radius = #.##
46 ostream &operator<<( ostream &output, const Circle &c )
47 {
48     output << "Center = " << static_cast< Point > ( c )
49         << "; Radius = "
50         << setiosflags( ios::fixed | ios::showpoint )
51         << setprecision( 2 ) << c.radius;
52
53     return output;    // enables cascaded calls
54 }
55 // Fig. 9.9: fig09_09.cpp
56 // Driver for class Circle
57 #include <iostream.h>
58 #include "point2.h"
59 #include "circle2.h"
60
61 int main()
62 {
63     Circle c( 2.5, 37, 43 );
64
65     cout << "X coordinate is " << c.getX()
66         << "\nY coordinate is " << c.getY()
67         << "\nRadius is " << c.getRadius();
68
69     c.setRadius( 4.25 );
70     c.setPoint( 2, 2 );
71     cout << "\n\nThe new location and radius of c are\n"
72         << c << "\nArea " << c.area() << '\n';
73
74     Point &pRef = c;
75     cout << "\nCircle printed as a Point is: " << pRef << endl;
76
77     return 0;
78 }

```

**输出结果:**

```

X coordinate is 37
Y coordinate is 43
Radius is 2.5

```

```
The new location and radius of c are
Center = [ 2,2]; Radius = 4.25
Area 56.74
```

```
Circle printed as a Point is: [ 2, 2]
```

图 9.9 演示 Circle 类型

最后一个例子是图 9.10。该例复用了类 Point 和类 Circle 的定义以及图 9.8 和图 9.9 中的成员函数的定义，分别是类 Cylinder 的定义、Cylinder 成员函数的定义以及类的驱动程序。类 Cylinder 对类 Circle 的继承是 public 继承，这意味着类 Cylinder 的 public 接口包括类 Circle 的成员函数、类 Point 的成员函数以及成员函数 setHeight、getHeight、area（对 Circle 中的 area 重新定义）以及 volume。注意 Cylinder 构造函数要调用直接基类 Circle 的构造函数，而不调用间接基类 Point 的构造函数。每个派生类构造函数只负责调用直接基类的构造函数（多重继承中可能有多个直接基类）。另外，注意 Cylinder 重载 operator<< 函数，Cylinder 类的友元可以通过将 Cylinder 引用 c 强制转换为 Circle 而输出 Cylinder 的 Circle 部分。因此调用 Circle 的 operator<< 并用相应 Circle 格式输出 x 和 y 坐标。

驱动程序实例化了类 Cylinder 的一个对象，然后用“get”函数读取该对象的信息。main 函数既不是类 Cylinder 的成员函数也不是其友元，因此它不能直接引用类 Cylinder 的 protected 数据。为此，程序用“set”函数 setHeight 和 setRadius 以及 setPoint 重新设置圆柱体的高度、半径和坐标值。最后，驱动程序先把 Point &（对 Point 对象的引用）类型的引用变量 pRef 初始化为类 Cylinder 的对象 cyl，然后打印出 pRef，尽管它已经被初始化为一个 Cylinder 对象，但是它还是“认为”自己是一个 Point 对象，所以该 Cylinder 对象实际上作为一个 Point 对象来打印；其次，将 Circle &（对 Circle 的引用）类型的引用变量 cRef 初始化为 Cylinder 对象 cyl，然后驱动程序打印 circleRef，尽管它已经被初始化为一个 Cylinder 对象，它还“认为”自己是一个 Circle 对象，所以该 Cylinder 对象实际上是作为一个 Circle 对象打印的，Circle 的面积也同时打印出来。

这个例子很好地演示了 public 继承以及对 protected 数据成员的定义和引用，读者现在应该对继承的基本知识有了一定的了解。下一章要讨论如何用多态性编写具有继承层次结构的程序。数据抽象、继承和多态性是面向对象程序设计的关键所在。

```
1 // Fig. 9.10: cylindr2.h
2 // Definition of class Cylinder
3 #ifndef CYLINDR2_H
4 #define CYLINDR2_H
5
6 #include "circle2.h"
7
8 class Cylinder : public Circle {
9     friend ostream &operator<<( ostream &, const Cylinder & );
10
11 public:
12     // default constructor
13     Cylinder( double h = 0.0, double r = 0.0,
14             int x = 0, int y = 0 );
15
16     void setHeight( double );    // set height
17     double getHeight() const;    // return height
18     double area() const;        // calculate and return area
```

```
19 double volume() const;      // calculate and return volume
20
21 protected:
22     double height;           // height of the Cylinder
23 };
24
25 #endif
26 // Fig. 9.10: cylindr2.cpp
27 // Member and friend function definitions
28 // for class Cylinder.
29 #include <iostream.h>
30 #include <iomanip.h>
31 #include "cylindr2.h"
32
33 // Cylinder constructor calls Circle constructor
34 Cylinder::Cylinder( double h, double r, int x, int y )
35     : Circle( r, x, y )      // call base-class constructor
36 { setHeight( h ); }
37
38 // Set height of Cylinder
39 void Cylinder::setHeight( double h )
40     { height = ( h >= 0 ? h : 0 ); }
41
42 // Get height of Cylinder
43 double Cylinder::getHeight() const { return height; }
44
45 // Calculate area of Cylinder (i.e., surface area)
46 double Cylinder::area() const
47 {
48     return 2 * Circle::area() +
49           2 * 3.14159 * radius * height;
50 }
51
52 // Calculate volume of Cylinder
53 double Cylinder::volume() const
54     { return Circle::area() * height; }
55
56 // Output Cylinder dimensions
57 ostream &operator<<( ostream &output, const Cylinder &c )
58 {
59     output << static_cast< Circle >( c )
60           << "; Height = " << c.height;
61
62     return output;    // enables cascaded calls
63 }
64 // Fig. 9.10: fig09_10.cpp
65 // Driver for class Cylinder
66 #include <iostream.h>
67 #include <iomanip.h>
68 #include "point2.h"
69 #include "circle2.h"
70 #include "cylindr2.h"
71
72 int main()
73 {
```

```
74 // create Cylinder object
75 Cylinder cyl( 5.7, 2.5, 12, 23 );
76
77 // use get functions to display the Cylinder
78 cout << "X coordinate is " << cyl.getX()
79     << "\nY coordinate is " << cyl.getY()
80     << "\nRadius is " << cyl.getRadius()
81     << "\nHeight is " << cyl.getHeight() << "\n\n";
82
83 // use set functions to change the Cylinder's attributes
84 cyl.setHeight( 10 );
85 cyl.setRadius( 4.25 );
86 cyl.setPoint( 2, 2 );
87 cout << "The new location, radius, and height of cyl are:\n"
88     << cyl << '\n';
89
90 // display the Cylinder as a Point
91 Point &pRef = cyl; // pRef "thinks" it is a Point
92 cout << "\nCylinder printed as a Point is: "
93     << pRef << "\n\n";
94
95 // display the Cylinder as a Circle
96 Circle &circleRef = cyl; // circleRef thinks it is a Circle
97 cout << "Cylinder printed as a Circle is:\n" << circleRef
98     << "\nArea: " << circleRef.area() << endl;
99
100 return 0;
101 }
```

**输出结果:**

```
X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7
```

```
The new location, radius, and height of cyl are:
Center = [ 2, 2]; Radius = 4.25; Height = 10.00
```

```
Cylinder printed as a Point is: [ 2, 2]
```

```
Cylinder printed as a Circle is:
Center = [ 2, 2]; Radius = 4.25
Area: 56.74
```

图 9.10 演示 Cylinder 类

## 9.15 多重继承

本章前面讨论了单一继承,即一个类是从一个基类派生来的。一个类也可以从多个基类派生而来,这种派生称为“多重继承”(multiple inheritance)。多重继承意味着一个派生类可以继承多个基类的成员,这种强大的功能支持了软件的复用性,但可能会引起大量的歧义性问题。



## 编程技巧 9.1

多重继承使用得好可具有强大的功能。当新类型与两个或多个现有类型之间存在“是”关系时（即类型 A “是”类型 B 并且也“是”类型 C）应该使用多重继承。

图 9.11 中的程序是一个多重继承的例子。类 Base1 包含一个 protected 数据成员 int value，还包含设置 value 值的构造函数和返回 value 值的 public 成员函数 getData。

类 Base2 和类 Base1 相似，只不过它的 protected 数据成员是 char letter。类 Base2 也包含一个 public 成员函数 getData，但是该函数返回的是 char letter 的值。

类 Derived 通过多重继承机制继承了类 Base1 和类 Base2，它有一个 private 数据成员 float real 和一个读取 float real 的 public 成员函数 getReal。

多重继承是非常直接的，即在 class derived 后的冒号(:)之后跟上用逗号分开的公有基类列表。还可以看到，构造函数 Derived 显式地调用了每个基类（即 Base1 和 Base2）的构造函数。同样，按指定的继承顺序调用基类构造函数，而不是按构造函数出现的顺序调用。如果成员初始化值列表中不显式调用基类构造函数，则隐式调用基类的默认构造函数。

```

1 // Fig. 9.11: basel.h
2 // Definition of class Base1
3 #ifndef BASE1_H
4 #define BASE1_H
5
6 class Base1 {
7 public:
8     Base1( int x ) { value = x; }
9     int getData() const { return value; }
10 protected:      // accessible to derived classes
11     int value;    // inherited by derived class
12 };
13
14 #endif
15 // Fig. 9.11: base2.h
16 // Definition of class Base2
17 #ifndef BASE2_H
18 #define BASE2_H
19
20 class Base2 {
21 public:
22     Base2( char c ) { letter = c; }
23     char getData() const { return letter; }
24 protected:      // accessible to derived classes
25     char letter;  // inherited by derived class
26 };
27
28 #endif
29 // Fig. 9.11: derived.h
30 // Definition of class Derived which inherits
31 // multiple base classes (Base1 and Base2).
32 #ifndef DERIVED_H
33 #define DERIVED_H
34
35 #include "basel.h"
36 #include "base2.h"

```

```

37
38 // multiple inheritance
39 class Derived : public Base1, public Base2 {
40     friend ostream &operator<<( ostream &, const Derived & );
41
42 public:
43     Derived( int, char, double );
44     double getReal() const;
45
46 private:
47     double real;    // derived class's private data
48 };
49
50 #endif
51 // Fig. 9.11: derived.cpp
52 // Member function definitions for class Derived
53 #include <iostream.h>
54 #include "derived.h"
55
56 // Constructor for Derived calls constructors for
57 // class Base1 and class Base2.
58 // Use member initializers to call base-class constructors
59 Derived::Derived( int i, char c, double f )
60     : Base1( i ), Base2( c ), real ( f ) { }
61
62 // Return the value of real
63 double Derived::getReal() const { return real; }
64
65 // Display all the data members of Derived
66 ostream &operator<<( ostream &output, const Derived &d )
67 {
68     output << "    Integer: " << d.value
69         << "\n Character: " << d.letter
70         << "\nReal number: " << d.real;
71
72     return output;    // enables cascaded calls
73 }
74 // Fig. 9.11: fig09_11.cpp
75 // Driver for multiple inheritance example
76 #include <iostream.h>
77 #include "base1.h"
78 #include "base2.h"
79 #include "derived.h"
80
81 int main()
82 {
83     Base1 b1( 10 ), *base1Ptr = 0; // create Base1 object
84     Base2 b2( 'Z' ), *base2Ptr = 0; // create Base2 object
85     Derived d( 7, 'A', 3.5 );      // create Derived object
86
87     // print data members of base class objects
88     cout << "Object b1 contains integer " << b1.getData()
89         << "\nObject b2 contains character " << b2.getData()
90         << "\nObject d contains:\n" << d << "\n\n";
91

```

```

92 // print data members of derived class object
93 // scope resolution operator resolves getData ambiguity
94 cout << "Data members of Derived can be"
95 << " accessed individually:"
96 << "\n Integer: " << d.Base1::getData()
97 << "\n Character: " << d.Base2::getData()
98 << "\nReal number: " << d.getReal() << "\n\n";
99
100 cout << "Derived can be treated as an "
101 << "object of either base class:\n";
102
103 // treat Derived as a Base1 object
104 base1Ptr = &d;
105 cout << "base1Ptr->getData() yields "
106 << base1Ptr->getData() << '\n';
107
108 // treat Derived as a Base2 object
109 base2Ptr = &d;
110 cout << "base2Ptr->getData() yields "
111 << base2Ptr->getData() << endl;
112
113 return 0;
114 }

```

**输出结果:**

```

object b1 contains integer 10
object b2 contains character z
object d contains:
    Integer: 7
    Character: A
    Real number:3.5

```

```

Data members of Derived can be accessed individually:
    Integer: 7
    Character: A
    Real number:3.5

```

```

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A

```

图 9.11 多重继承的程序

在 Derived 中重载的流插入运算符通过派生类对象 d 用圆点表示法打印 value、letter 和 real 的值。因为该运算符函数是类 Derived 的友元，所以 operator<< 可以直接访问类 Derived 的 private 数据成员 real，还能访问 Base1 和 Base2 的 protected 数据成员 value 和 letter。

下面探讨一下 main 函数中的驱动程序。程序中首先建立了类 Base1 的对象 b1 和类 Base2 的对象 b2，并将它们分别初始化为 int 类型的值 10 和 char 类型的值 'z'，然后建立类 Derived 的对象 d 并将其初始化成包括 int 类型的值 7、char 类型的值 'A' 和 float 类型的值 3.5。

通过调用每个对象的 getData 函数打印每个基类对象的内容。尽管有两种 getData 函数，但是因为直接引用了对象 b1 和 b2 的 getData 函数版本，因此对它们的调用没有歧义性问题。

接下来用静态关联打印出 Derived 的对象 d 的内容。因为该对象包含两个 getData 函数，一个是从类 Base1 继承来的，另一个是从 Base2 继承来的，所以存在着歧义性问题。用二元作用域运算符很容易解决这个问题。例如，d.Base1::getData() 打印了 int 类型的 value 值，d.Base2::getData() 打印了 char 类型的 letter 值。调用 d.getReal() 打印 float 类型的 real 值不存在歧义性问题。然后演示了单一继承的“是”关系也适用于多重继承。程序中把派生类对象 d 的地址赋给了基类指针 Base1Ptr，并用该指针调用 Base1 的成员函数 getData 打印出 int value 值。之后又把 d 的地址赋给基类指针 Base2Ptr，并用该指针调用 Base2 的成员函数 getData 打印出 char letter 的值。

这个例子展示了多重继承的机制并介绍了一种简单的歧义性问题。多重继承是一个很复杂的话题，许多高级 C++ 书籍对此有详细的论述。

#### 软件工程视点 9.12

多重继承是个强大的功能，但可能增加系统的复杂性。使用多重继承的系统需要更加认真设计，能用单一继承时应尽量使用单一继承。

## 小结

- 面向对象的程序设计能力的关键之一是通过继承实现软件的复用。
- 程序员可以让新类继承已定义基类的数据成员和成员函数，而不必重新编写新的数据成员和成员函数。这种新类称为派生类。
- 对于单一继承，派生类只有一个基类。对于多重继承，派生类常常是从多个基类派生出来的，这些基类之间可能毫无关系。
- 派生类通常添加了其自身的数据成员和成员函数，因而通常比基类大得多。派生类比基类更具体，它代表了一组外延较小的对象。
- 派生类不能访问其基类的 private 成员，否则会破坏基类的封装性。但是，派生类能够访问基类的 public 成员和 protected 成员。
- 派生类的构造函数总是先调用其基类构造函数来初始化派生类中的基类成员。
- 调用析构函数的次序和调用构造函数的次序相反，因此派生类析构函数在基类析构函数之前调用。
- 利用继承能够实现软件复用。软件复用缩短了程序的开发时间，促使开发人员复用已经测试和调试好的高质量的软件。
- 可以用现有的类库实现继承。
- 将来有一天，软件也可以像如今的硬件一样用标准的可复用组件进行构造。
- 派生类不必访问基类的源代码，但是需要知道基类的接口和能够连接到基类的目标代码。
- 派生类对象可作为其 public 基类的对象处理，但是反过来不行。
- 基类和派生类之间具有层次关系。
- 一个类可以单独存在，但是当利用继承机制使用该类时，该类就成为给其他类提供属性和行为的基类，或者成为继承其他类的属性和行为的派生类。
- 继承层次的深度在特定系统的限制范围之内是任意的。
- 层次结构是管理和弄清复杂问题的有用工具。随着软件日益复杂化，C++ 提供了支持层次结构的继承和多态性机制。

- 程序员可以用显式类型转换把基类指针转换为派生类指针。但是,如果要复引用该指针,那么在转换前首先应该把它指向某个派生类的对象。
- protected 访问是介于 public 访问和 private 访问之间的中间层次。基类的 protected 成员只能被基类的成员和友元以及派生类的成员和友元访问,没有其他函数能访问基类的 protected 成员。
- protected 成员能用来扩展对派生类的访问权限,而拒绝非类成员函数和非友元函数的访问权。
- 通过在派生类名后放置冒号并紧跟逗号分隔的基类列表可以用来指明多重继承。在派生类构造函数调用基类构造函数时使用成员初始化值的列表。
- 从基类派生一个类时,基类可被声明为 public、protected 或者 private。
- 从 public 基类派生一个类时,基类的 public 成员成为派生类的 public 成员,基类的 protected 成员成为派生类的 protected 成员。
- 从 protected 基类派生一个类时,基类的 public 和 protected 成员都成为派生类的 protected 成员。从 private 基类派生一个类时,基类的 public 和 protected 成员均成为派生类的 private 成员。
- 基类既可能是派生类的直接基类,也可能是派生类的间接基类。在声明派生类时,派生类的首部要显式地列出直接基类。间接基类不是显式地列在派生类的头部,而是沿着类的多个层次向上继承。
- 不适合派生类的基类成员可以在派生类中重新定义。
- 区分“是”关系和“有”关系是很重要的。在“是”关系中,派生类的对象也以作为基类的对象处理。而“有”关系中,一个类的对象拥有作为其成员的其他类的对象。“是”关系是一种继承,“有”关系是一种复合。
- 派生类对象可以赋给基类对象。由于派生类具有每一个基类成员,这种赋值是有意义的。
- 指向派生类对象的指针可以隐式地转换为指向基类对象的指针。
- 使用显式强制转换可以将基类指针转换为派生类指针,指向的目标应该是派生类的对象。
- 基类描述了共性。所有从基类派生出来的类都继承了基类的功能。在面向对象的设计过程中,设计者先寻求并提取出构成所需基类的共性,然后再通过继承从基类派生出超出基类功能的定制派生类。
- 因为派生类中没有列出所有的成员,所以浏览一组派生类的声明会令人迷惑,当派生类的声明中没有列出继承来的成员时更是如此,但是派生类中确实存在继承来的成员。
- “有”关系通过复合现有的类建立了新类。
- “知道”关系是对象包含其他对象的指针或引用,因而知道存在这些对象。
- 成员对象构造函数以声明对象的顺序调用。对继承关系而言,基类构造函数以指定的继承顺序调用并且在调用派生类构造函数之前调用。
- 对于一个派生类对象,先调用基类的构造函数,然后调用派生类的构造函数(也许调用成员对象的构造函数)。
- 当取消派生类对象时,析构函数的调用顺序与调用构造函数的顺序相反,即先调用派生类的析构函数,然后调用基类的析构函数。
- 一个类可以从多个基类派生而来,这种派生称为多重继承。
- 在继承指示符即冒号(:)后跟上用逗号分开的基类列表表示了多重继承。
- 派生类构造函数用成员初始化值语法调用其每个基类的构造函数。基类构造函数按其在继承中声明的基类顺序依次调用。

## 术语

|                                            |                                                 |
|--------------------------------------------|-------------------------------------------------|
| abstraction 抽象                             | inheritance 继承                                  |
| ambiguity in multiple inheritance 多重继承的歧义性 | is a relationship “是”关系                         |
| association 关联                             | knows a relationship “知道”关系                     |
| base class 基类                              | member access control 成员访问控制                    |
| base class default constructor 基类默认构造函数    | member class 成员类                                |
| base-class constructor 基类构造函数              | member object 成员对象                              |
| base-class destructor 基类析构函数               | multiple inheritance 多重继承                       |
| base-class initializer 基类初始化值              | object-oriented programming ( OOP ) 面向对象编程      |
| base-class pointer 基类指针                    | override a base-class member function 重定义基类成员函数 |
| class hierarchy 类层次                        | pointer to a base-class object 基类对象指针           |
| class libraries 类库                         | pointer to a derived-class object 派生类对象指针       |
| client of a class 类客户                      | private base class private 基类                   |
| composition 复合                             | private inheritance private 继承                  |
| customize software 定制的软件                   | protected base class protected 基类               |
| derived class 派生类                          | protected inheritance protected 继承              |
| derived-class constructor 派生类构造函数          | protected keyword protected 关键字                 |
| derived-class destructor 派生类析构函数           | protected member of a class 类的 protected 成员     |
| derived-class pointer 派生类指针                | public base class public 基类                     |
| direct base class 直接基类                     | public inheritance public 继承                    |
| friend of a base class 基类友元                | single inheritance 单一继承                         |
| friend of a derived class 派生类友元            | software reusability 软件复用性                      |
| function overriding 函数重定义                  | standardized software components 标准化软件组件        |
| has a relationship “是”关系                   | subclass 子类                                     |
| hierarchical relationship 层次关系             | superclass 超类                                   |
| indirect base class 间接基类                   | uses a relationship “使用”关系                      |
| infinite recursion error 无穷递归错误            |                                                 |

## 自测练习

### 9.1 填空

- 如果类 Alpha 继承了类 Beta, 则类 Alpha 称为 \_\_\_\_\_ 类, 类 Beta 称为 \_\_\_\_\_ 类。
- C++ 提供的 \_\_\_\_\_ 机制允许一个派生类继承多个基类, 即使这些基类是相互无关的。
- 利用继承能够实现 \_\_\_\_\_。这种实现缩短了程序的开发时间, 促使开发人员复用已经测试和调试好的高质量软件。
- \_\_\_\_\_ 类的对象可作为 \_\_\_\_\_ 类的对象处理。
- 为了将基类指针转换为派生类指针, 由于编译器认为这种操作是危险的, 所以要使用 \_\_\_\_\_。
- 三种成员访问说明符分别是 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。

- g) 当用 public 继承从基类派生一个类时, 基类的 public 成员成为派生类的 \_\_\_\_\_ 成员, protected 成员成为派生类的 \_\_\_\_\_ 成员。
- h) 当用 protected 继承从基类派生一个类时, 基类的 public 成员成为派生类的 \_\_\_\_\_ 成员, 基类的 protected 成员成为派生类的 \_\_\_\_\_ 成员。
- i) 类之间的“有”关系代表 \_\_\_\_\_, “是”关系表示 \_\_\_\_\_。

### 自测练习答案

- 9.1 a) 派生、基。b) 多重继承。c) 软件复用。d) 派生、基。e) 强制类型转换。f) public、protected、private。g) public、protected。h) protected、protected。i) 复合、继承。

### 练习

- 9.2 考虑类 Bicycle, 根据你对自行车通用组件的了解, 描述类 Bicycle 继承其他类 (这些类还可以再继承其他类) 的层次结构。讨论类 Bicycle 的各种对象的实例以及其他紧密相关的派生类对类 Bicycle 的继承性。
- 9.3 简要地定义下列术语: 继承、多重继承、基类、派生类。
- 9.4 为什么编译器认为把基类指针转换为派生类指针是危险的?
- 9.5 区分单一继承与多重继承。
- 9.6 (判断对错) 派生类通常称为子类, 因为它表示基类的子集, 即派生类通常比基类更小。
- 9.7 (判断对错) 派生类对象也是其基类的对象。
- 9.8 有些程序员不喜欢用 protected 访问, 因为它会破坏基类的封装。试讨论使用 protected 访问与坚持在基类中使用 private 访问的利与弊。
- 9.9 许多带继承的程序可以改用复合, 反过来也成立。试讨论这两种方法在本章 Point、Circle、Cylinder 类继承中的利弊。将图 9.10 (及其支持类) 改写为使用复合而不用继承。然后重新评估这两种方法对 Point、Circle、Cylinder 问题和一般面向对象程序的利弊。
- 9.10 将图 9.10 的 Point、Circle、Cylinder 程序改写为 Point、Square、Cube 程序。用两种方法, 一种用继承, 一种用复合。
- 9.11 本章曾经指出, “基类成员不适合派生类时, 可以在派生类中重定义该成员的实现方法”。如果这样, “派生类对象是基类对象”的关系是否仍然成立? 为什么。
- 9.12 研究图 9.2 的继承层次。对每个类, 指出该层次中的共同属性与行为。并增加一些类以丰富这个层次 (如 UndergraduateStudent、GraduateStudent、Freshman、Sophomore、Junior、Senior 等等)。
- 9.13 写出 Quadrilateral、Trapezoid、Parallelogram、Rectangle 和 Square 类的继承层次。用 Quadrilateral 作为这个层次的基类。让层次尽量深 (尽量多层)。Quadrilateral 的 private 数据为 Quadrilateral 四角的 (x, y) 坐标对。编写一个驱动程序, 实例化和显示这些类的对象。
- 9.14 试写出所能想到的所有形状 (包括二维的和三维的), 并生成一个形状层次结构。生成的层次结构要以 Shape 作为基类, 并由此派生出类 TwoDimensionalShape 和 ThreeDimensionalShape。开发出层次结构后, 定义其中的每一个类。第 10 章的练习要用这个层次结构处理作为基类 Shape 的对象的形状, 这种技术叫做多态性。

## 第10章 虚函数和多态性

### 教学目标

- 了解多态性的概念
- 了解怎样声明和使用实现多态性的虚函数
- 了解抽象类和具体类的区别
- 学会怎样声明建立抽象类的纯虚函数
- 认识多态性是如何扩展和维护系统
- 了解 C++ 如何实现虚函数和动态关联

### 10.1 简介

虚函数 (virtual function) 和多态性 (polymorphism) 使得设计和实现易于扩展的系统成为可能。程序可以对层次中所有现有类的对象 (基类对象) 进行一般性处理。程序开发期间不存在的类可以用一般化程序稍作修改或不经修改即加进去, 只要这些类属于一般处理的继承层次。程序中惟一要修改的部分是需要直接了解加进层次中的特定类的部分。

### 10.2 类型域和 switch 语句

处理多种不同类型对象的手段之一是使用 switch 语句。switch 语句能够根据每一种对象的类型选择对该对象合适的操作。例如, 在形状层次中, 每个形状指定自己的类型数据成员, switch 结构可以根据特定对象的类型确定调用哪个 print 函数。

但是, 使用 switch 逻辑存在许多问题。例如, 程序员可能会忘记应有的类型测试; 在一条 switch 语句中可能会忘记测试所有可能的情况; 在修改基于 switch 语句的系统时可能会忘记在现有的 switch 语句中插入新类; 为了处理新的类型, 每次修改 switch 语句都要修改系统中的每一条 switch 语句, 这很费时并且容易出错。

正如以后会看到的, 利用了虚函数和多态性的程序设计无需使用 switch 逻辑。程序员可以用虚函数机制自动完成等价的逻辑, 因而避免与 switch 逻辑有关的各种各样的错误。

#### 软件工程视点 10.1

使用虚函数和多态性可简化源代码的长度。为支持更简单的顺序代码, 虚函数和多态性包含的分支逻辑更少。这种简化有助于程序的测试、调试和维护。

### 10.3 虚函数

假定一组形状类 (如 Circle、Triangle、Rectangle 和 Square 等等) 都是从基类 Shape 派生出来的。在面向对象的程序设计中, 我们可能要使每一个这样的类都能够绘制其自身形状。尽管每个类



都有它自己 draw 函数,但是绘制每种形状的 draw 函数却是大不相同的。当需要绘制形状时,不管它是什么形状,把它作为基类 Shape 的对象处理是再好不过的。然后,我们只需要简单地调用基类 Shape 的函数 draw,并让程序动态地确定(即在运行时确定)使用哪个派生类的 draw 函数。

为了使这种行为可行,我们把基类中的函数 draw 声明为虚函数,然后在每个派生类中重新定义 draw 使之能够绘制合适的形状。虚函数的声明方法是在基类的函数原型前加上关键字 virtual。例如,基类 Shape 中可能出现:

```
virtual void draw() const;
```

上述原型声明函数 draw 是不取参数也不返回数值的常量函数,而且是个虚函数。

#### 软件工程视点 10.2

一旦一个函数被声明为虚函数,即使重新定义类时没有声明虚函数,那么它从该点之后的继承层次结构中都是虚函数。

#### 编程技巧 10.1

虽然函数在类层次结构的高层中声明为虚函数会使它在低层隐式地成为虚函数,但有些程序员为了提高程序的清晰性更喜欢在每一层中显式地声明这些虚函数。

#### 软件工程视点 10.3

没有定义虚函数的派生类简单地继承其直接基类的虚函数。

如果在基类中将函数 draw 声明为 virtual,然后用基类指针或引用指明派生类对象并使用该指针调用 draw 函数(如 shapePtr->draw()),则程序会动态地(即在运行时)选择该派生类的 draw 函数,这称为动态关联(见 10.6 和 10.9 节的实例研究)。

如果用名字和圆点成员选择运算符引用一个特定的对象来调用虚函数(如 squareObject.draw()),则被调用虚函数是在编译时确定的(称为静态关联),调用的虚函数也就是为该特定对象的类定义(或继承该特定对象类)的函数。

## 10.4 抽象基类和具体类

当我们把类看作一种数据类型时,我们通常认定该类型的对象是要被实例化的。但是,在许多情况下,定义不实例化为任何对象的类是很有用处的,这种类称为“抽象类”(abstract class)。因为抽象类要作为基类被其他类继承,所以通常也把它称为“抽象基类”(abstract base class)。抽象基类不能用来建立实例化的对象。

抽象类的惟一用途是为其他类提供合适的基类,其他类可从它这里继承和(或)实现接口。能够建立实例化对象的类称为具体类(concrete class)。

例如,我们可以建立抽象基类 TwoDimensionalObject,然后从它派生出具体类 Square、Circle 和 Triangle 等等,也可以建立抽象基类 ThreeDimensionalObject,然后从它派生出具体类 Cube、Sphere 和 Cylinder 等等。这些抽象基类表述的含义因为太广泛而定义不出实在的对象。如果要建立实例对象,则需要含义更加明确的类,这就是所谓的“具体类”。具体类具有足以能够建立实例化对象的明确含义。

如果将带有虚函数的类中的一个或者多个虚函数声明为纯虚函数,则该类就成为抽象类。纯虚函数是在声明时“初始化值”为 0 的函数,例如:

```
virtual float earnings() const = 0;    // pure virtual
```

#### 软件工程视点 10.4

如果某个类是从一个带有纯虚函数的类派生出来的, 并且没有在该派生类中提供该纯虚函数的定义, 则该虚函数在派生类中仍然是纯虚函数, 因而该派生类也是一个抽象类。

#### 常见编程错误 10.1

试图实例化一个抽象类对象 (即包含一个或者多个纯虚函数的类) 是一种语法错误。

一个类层次结构中可以不包含任何抽象类, 但是正如以后会看到的, 很多良好的面向对象的系统, 其类层次结构的顶部是一个抽象基类。在有些情况中, 类层次结构顶部有好几层都是抽象类。形状类的层次结构就是一种典型的范例。我们可以在该层次结构的顶部建立抽象基类 Shape, 在往下一层中还可以再建立两个抽象基类, 即二维形状类 TwoDimensionalShape 和三维形状类 ThreeDimensionalShape, 再往下我们就可以开始定义二维形状的具体类如圆形类和正方形类以及三维形状的具体类如球类和立方体类等等。

## 10.5 多态性

C++ 支持多态性。所谓多态性是指: 通过继承相关的不同的类, 他们的对象能够对同一个函数调用作出不同的响应。例如, 如果类 Rectangle 是从类 Quadrilateral 派生出来的, 那么类 Rectangle 的对象比类 Quadrilateral 的对象的具体, 对类 Quadrilateral 的操作 (如计算周长和面积) 也能用在类 Rectangle 的对象上。

多态性是通过虚函数实现的。当通过基类指针 (或引用) 请求使用虚函数时, C++ 会在与对象关联的派生类中正确地选择重定义的函数。

有时候在基类中定义的非虚函数会在派生类中重新定义。如果用基类指针调用该成员函数, 则选择基类版本的成员函数; 如果用派生类指针调用该成员函数, 则选择派生类版本的成员函数。这不是多态性行为。

下面的例子使用图 9.5 的基类 Employee 和派生类 HourlyWorker:

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print();           // call base-class print function
hPtr-> print();           // call derived-class print function
ePtr = &h;               // allowable implicit conversion
ePtr->print();           // still calls base-class print
```

基类 Employee 和派生类 HourlyWorker 都定义了自己的 print 函数。由于这个函数没有声明为 virtual, 而且签名相同, 因此通过 Employee 指针调用 print 函数时调用 Employee::print() (不管 Employee 指针指向基类对象还是派生类 HourlyWorker 对象), 而通过 HourlyWorker 指针调用 print 函数则调用 Worker::print()。派生类也可以调用基类函数, 但派生类对象通过派生类对象的指针调用基类 print 时, 函数要显式调用如下:

```
hPtr-> Employee::print();           // call base-class print function
```

表示调用基类 print。

使用虚函数和多态性能够使成员函数的调用根据接收到该调用的对象的类型产生不同的动作（但会需要少量执行时的开销）多态性赋予了程序员极大的灵活性。下面几节要举例说明多态性和虚函数的功能。

#### 软件工程视点 10.5

利用虚函数和多态性，程序员可以处理普遍性而让执行环境处理特殊性。即使在不知道一些对象的类型的情况下，程序员也可以命令各种各样的对象表现出适合这些对象的行为。

#### 软件工程视点 10.6

多态性提高了可扩展性：处理多态性行为的软件可以用与接收消息的对象类型无关的方式编写。因此，不必修改基本系统就可以把能够响应现有消息的新类型的对象添加到系统中。除了实例化新对象的客户代码需要重新编译外，程序无需重新编译。

#### 软件工程视点 10.7

抽象类为类层次结构中的各个成员定义接口。抽象类中包含了要在派生类中定义的纯虚函数，该层次结构中的所有函数都可以通过多态性使用同样的接口。

尽管不能实例化抽象基类的对象，但是可以声明引用抽象基类的指针。当实例化了具体类的对象后，可以用这种指针使派生类对象具有多态操作能力。

下面考虑一个应用多态性和虚函数的例子。一个屏幕管理程序需要显示各种各样的对象，甚至包括在屏幕管理程序编写后又添加到系统中的新类型的对象。系统可能需要显示各种各样的形状，例如正方形、圆形、三角形、矩形等等（每一个类都是基类 Shape 的派生类）。屏幕管理程序使用基类指针（指向 Shape）来管理要显示的对象。为了能够绘制所有的对象（不管该对象在继承层次结构中的哪一层），管理程序都是使用指向该对象的基类指针并向该对象简单地发送一条 draw 消息。函数 draw 在基类 Shape 中被声明为纯虚函数，并且在每一个派生类中被重新定义，每个对象都知道如何绘制自身。屏幕管理程序不必关心这些细节内容，它只要简单地告诉每个对象进行绘制即可。

多态性特别适合于实现分层的软件系统。例如，在操作系统中各种类型的物理设备彼此之间的操作是不同的，然而从设备读取数据和把数据写入设备的命令在某种程度是统一的。发送给设备驱动程序对象的“写”消息（write 函数调用）需要在该设备驱动程序的上下文中具体地解释，并且还要解释设备驱动程序是如何操作该特定类型设备的。但是，write 调用本身和对任何其他对象的 write 调用实际上没有什么区别，都只是把内存中一定数目的字节放在设备中。面向对象的操作系统可能会用抽象基类为所有设备驱动程序提供合适的接口，然后通过继承抽象基类生成执行所有类似操作的派生类。设备驱动程序所提供的功能（即 public 接口）在抽象基类中则是以纯虚函数形式出现的，派生类中提供了这些虚函数的实现，已实现的函数能够响应特定类型的设备驱动程序。

利用多态编程，程序可以从类层次的不同层中遍历对象的指针数组。这种数组中的指针都是派生类对象的基类指针。例如，TwoDimensionalShape 类的对象数组可以包含指向派生类 Square、Circle、Triangle、Rectangle 和 Line 等对象的 TwoDimensionalShape\* 指针。使用多态编程时，发出一个绘制数组中每个对象的消息即可在屏幕上画出正确的图形。

## 10.6 实例研究：利用多态性的工资单系统

下面的范例程序用虚函数和多态性根据雇员的类型完成工资单的计算（见图 10.1）。所用的基类是雇员类 Employee，其派生类包括：老板类 Boss，不管工作多长时间他总是有固定的周薪；销售

员类 `CommissionWorker`，他的收入是一小部分基本工资加上销售额的一定的百分比；计件工类 `PieceworkWorker`，他的收入取决他生产的工件数量；小时工类 `HourlyWorker`，他的收入以小时计算，再加上加班费。

函数 `earnings` 的调用当然要普遍适用于所有的雇员。每人收入的计算方法取决于它属于哪一类雇员。因为这些类都是由基类 `Employee` 派生出来的，所以函数 `earnings` 在基类 `Employee` 中被声明为 `virtual`，并在每个派生类中都正确地实现 `earnings`。为计算任何雇员的收入，程序简单地使用了一个指向该雇员对象的基类指针并调用函数 `earnings`。在一个实际的工资单系统中，各种雇员对象可能保存在一个数组（链表）中，数组每个指针都是 `Employee *` 类型，然后程序遍历链表中的每一个节点，并在每一个节点处用 `Employee *` 指针调用对象的 `earnings` 函数。

下面看一看类 `Employee`。该类的 `public` 成员函数包括：构造函数，该构造函数有两个参数，第一个参数是雇员的姓，第二个参数是雇员的名；析构函数，用来释放动态分配的内存；两个“`get`”函数，分别返回雇员的姓和名；纯虚函数 `earnings` 和虚函数 `print`。为什么要把 `earnings` 函数声明为纯虚函数呢？因为在类 `Employee` 中提供这个函数的实现是没有意义的，将它声明为纯虚函数表示要在派生类中而不是在基类中提供具体的实现。对于具有广泛含义的雇员，我们不能计算出他的收入，而必须首先知道该雇员的类型。程序员不会试图在基类 `Employee` 中调用该纯虚函数，所有的派生类根据相应的实现为这些类重定义 `earnings`。

类 `Boss` 是通过 `public` 继承从类 `Employee` 派生出来的，它的 `public` 成员函数包括：构造函数，构造函数有三个参数，即雇员的姓和名以及周薪，为了初始化派生类对象中基类部分的成员 `firstName` 和 `lastName`，雇员的姓和名传递给了类 `Employee` 的构造函数；“`set`”函数，用来把新值赋给 `private` 数据成员 `weeklySalary`；虚函数 `earnings`，用来定义如何计算 `Boss` 的工资；虚函数 `print`，它输出雇员类型，然后调用 `Employee::print()` 输出员工姓名。

类 `CommissionWorker` 是通过 `public` 继承从类 `Employee` 派生出的，它的 `public` 成员函数包括：构造函数，构造函数有五个参数，即姓、名、基本工资、回扣及产品销售量，并将姓和名传递给了类 `Employee` 的构造函数；“`set`”函数，用于将新值赋给 `private` 数据成员 `salary`、`commission` 和 `quantity`；虚函数 `earnings`，用来定义如何计算 `CommissionWorker` 的工资；虚函数 `print`，输出雇员类型，然后调用 `Employee::print()` 输出员工姓名。

类 `PieceWorker` 是通过 `public` 继承从类 `Employee` 派生出来的，`public` 成员函数包括：构造函数，构造函数有四个参数，即计件工的姓、名、每件产品的工资以及生产的产品数量，并将姓和名传递给了类 `Employee` 的构造函数；“`set`”函数，用来将新值赋给 `private` 数据成员 `wagePerPiece` 和 `quantity`；虚函数 `earnings`，用来定义如何计算 `PieceWorker` 的工资；虚函数 `print`，它输出雇员类型，然后调用 `Employee::print()` 输出员工姓名。

类 `HourlyWorker` 是通过 `public` 继承从类 `Employee` 派生出来的，`public` 成员函数包括：构造函数，构造函数有四个参数，即姓、名、每小时工资及工作的时间数，并将姓、名传递给了类 `Employee` 的构造函数；“`set`”函数，将新值赋给 `private` 数据成员 `wage` 和 `hours`；虚函数 `earnings`，用来定义如何计算 `HourlyWorker` 的工资；虚函数 `print`，输出雇员类型，然后调用 `Employee::print()` 输出员工姓名。

```
1 // Fig. 10.1: employ2.h
2 // Abstract base class Employee
3 #ifndef EMPLOY2_H
4 #define EMPLOY2_H
5
6 #include <iostream.h>
```

```
7
8 class Employee {
9 public:
10     Employee( const char *, const char * );
11     ~Employee(); // destructor reclaims memory
12     const char *getFirstName() const;
13     const char *getLastName() const;
14
15     // Pure virtual function makes Employee abstract base class
16     virtual double earnings() const = 0; // pure virtual
17     virtual void print() const;          // virtual
18 private:
19     char *firstName;
20     char *lastName;
21 };
22
23 #endif
24 // Fig. 10.1: employ2.cpp
25 // Member function definitions for
26 // abstract base class Employee.
27 // Note: No definitions given for pure virtual functions.
28 #include <string.h>
29 #include <assert.h>
30 #include "employ2.h"
31
32 // Constructor dynamically allocates space for the
33 // first and last name and uses strcpy to copy
34 // the first and last names into the object.
35 Employee::Employee( const char *first, const char *last )
36 {
37     firstName = new char[ strlen( first ) + 1 ];
38     assert( firstName != 0 ); // test that new worked
39     strcpy( firstName, first );
40
41     lastName = new char[ strlen( last ) + 1 ];
42     assert( lastName != 0 ); // test that new worked
43     strcpy( lastName, last );
44 }
45
46 // Destructor deallocates dynamically allocated memory
47 Employee::~Employee()
48 {
49     delete [] firstName;
50     delete [] lastName;
51 }
52
53 // Return a pointer to the first name
54 // Const return type prevents caller from modifying private
55 // data. Caller should copy returned string before destructor
56 // deletes dynamic storage to prevent undefined pointer.
57 const char *Employee::getFirstName() const
58 {
59     return firstName; // caller must delete memory
60 }
61
62 // Return a pointer to the last name
```

---

```

63 // Const return type prevents caller from modifying private
64 // data. Caller should copy returned string before destructor
65 // deletes dynamic storage to prevent undefined pointer.
66 const char *Employee::getLastName() const
67 {
68     return lastName;    // caller must delete memory
69 }
70
71 // Print the name of the Employee
72 void Employee::print() const
73 { cout << firstName << ' ' << lastName; }
74 // Fig. 10.1: boss1.h
75 // Boss class derived from Employee
76 #ifndef BOSSL_H
77 #define BOSSL_H
78 #include "employ2.h"
79
80 class Boss : public Employee {
81 public:
82     Boss( const char *, const char *, double = 0.0 );
83     void setWeeklySalary( double );
84     virtual double earnings() const;
85     virtual void print() const;
86 private:
87     double weeklySalary;
88 };
89
90 #endif
91 // Fig. 10.1: boss1.cpp
92 // Member function definitions for class Boss
93 #include "boss1.h"
94
95 // Constructor function for class Boss
96 Boss::Boss( const char *first, const char *last, double s )
97     : Employee( first, last ) // call base-class constructor
98 { setWeeklySalary( s ); }
99
100 // Set the Boss's salary
101 void Boss::setWeeklySalary( double s )
102 { weeklySalary = s > 0 ? s : 0; }
103
104 // Get the Boss's pay
105 double Boss::earnings() const { return weeklySalary; }
106
107 // Print the Boss's name
108 void Boss::print() const
109 {
110     cout << "\n          Boss: ";
111     Employee::print();
112 }
113 // Fig. 10.1: commis1.h
114 // CommissionWorker class derived from Employee
115 #ifndef COMMIS1_H
116 #define COMMIS1_H
117 #include "employ2.h"
118

```

```
119 class CommissionWorker : public Employee {
120 public:
121     CommissionWorker( const char *, const char *,
122                       double = 0.0, double = 0.0,
123                       int = 0 );
124     void setSalary( double );
125     void setCommission( double );
126     void setQuantity( int );
127     virtual double earnings() const;
128     virtual void print() const;
129 private:
130     double salary;           // base salary per week
131     double commission;      // amount per item sold
132     int quantity;           // total items sold for week
133 };
134
135 #endif
136 // Fig. 10.1: commis1.cpp
137 // Member function definitions for class CommissionWorker
138 #include <iostream.h>
139 #include "commis1.h"
140
141 // Constructor for class CommissionWorker
142 CommissionWorker::CommissionWorker( const char *first,
143                                     const char *last, double s, double c, int q )
144     : Employee( first, last ) // call base-class constructor
145 {
146     setSalary( s );
147     setCommission( c );
148     setQuantity( q );
149 }
150
151 // Set CommissionWorker's weekly base salary
152 void CommissionWorker::setSalary( double s )
153     { salary = s > 0 ? s : 0; }
154
155 // Set CommissionWorker's commission
156 void CommissionWorker::setCommission( double c )
157     { commission = c > 0 ? c : 0; }
158
159 // Set CommissionWorker's quantity sold
160 void CommissionWorker::setQuantity( int q )
161     { quantity = q > 0 ? q : 0; }
162
163 // Determine CommissionWorker's earnings
164 double CommissionWorker::earnings() const
165     { return salary + commission * quantity; }
166
167 // Print the CommissionWorker's name
168 void CommissionWorker::print() const
169 {
170     cout << "\nCommission worker: ";
171     Employee::print();
172 }
173 // Fig. 10.1: piecel.h
174 // PieceWorker class derived from Employee
```

```
175 #ifndef PIECE1_H
176 #define PIECE1_H
177 #include "employ2.h"
178
179 class PieceWorker : public Employee {
180 public:
181     PieceWorker( const char *, const char *,
182                 double = 0.0, int = 0);
183     void setWage( double );
184     void setQuantity( int );
185     virtual double earnings() const;
186     virtual void print() const;
187 private:
188     double wagePerPiece; // wage for each piece output
189     int quantity;        // output for week
190 };
191
192 #endif
193 // Fig. 10.1: piece1.cpp
194 // Member function definitions for class PieceWorker
195 #include <iostream.h>
196 #include "piece1.h"
197
198 // Constructor for class PieceWorker
199 PieceWorker::PieceWorker( const char *first, const char *last,
200                          double w, int q )
201     : Employee( first, last ) // call base-class constructor
202 {
203     setWage( w );
204     setQuantity( q );
205 }
206
207 // Set the wage
208 void PieceWorker::setWage( double w )
209 { wagePerPiece = w > 0 ? w : 0; }
210
211 // Set the number of items output
212 void PieceWorker::setQuantity( int q )
213 { quantity = q > 0 ? q : 0; }
214
215 // Determine the PieceWorker's earnings
216 double PieceWorker::earnings() const
217 { return quantity * wagePerPiece; }
218
219 // Print the PieceWorker's name
220 void PieceWorker::print() const
221 {
222     cout << "\n    Piece worker: ";
223     Employee::print();
224 }
225 // Fig. 10.1: hourly1.h
226 // Definition of class HourlyWorker
227 #ifndef HOURLY1_H
228 #define HOURLY1_H
229 #include "employ2.h"
230
```



```
231 class HourlyWorker : public Employee {
232 public:
233     HourlyWorker( const char *, const char *,
234                  double = 0.0, double = 0.0);
235     void setWage( double );
236     void setHours( double );
237     virtual double earnings() const;
238     virtual void print() const;
239 private:
240     double wage;    // wage per hour
241     double hours;   // hours worked for week
242 };
243
244 #endif
245 // Fig. 10.1: hourly1.cpp
246 // Member function definitions for class HourlyWorker
247 #include <iostream.h>
248 #include "hourly1.h"
249
250 // Constructor for class HourlyWorker
251 HourlyWorker::HourlyWorker( const char *first,
252                             const char *last,
253                             double w, double h )
254     : Employee( first, last )    // call base-class constructor
255 {
256     setWage( w );
257     setHours( h );
258 }
259
260 // Set the wage
261 void HourlyWorker::setWage( double w )
262     { wage = w > 0 ? w : 0; }
263
264 // Set the hours worked
265 void HourlyWorker::setHours( double h )
266     { hours = h >= 0 && h < 168 ? h : 0; }
267
268 // Get the HourlyWorker's pay
269 double HourlyWorker::earnings() const
270 {
271     if ( hours <= 40 ) // no overtime
272         return wage * hours;
273     else                // overtime is paid at wage * 1.5
274         return 40 * wage + ( hours - 40 ) * wage * 1.5;
275 }
276
277 // Print the HourlyWorker's name
278 void HourlyWorker::print() const
279 {
280     cout << "\n    Hourly worker: ";
281     Employee::print();
282 }
283 // Fig. 10.1: fig10_01.cpp
284 // Driver for Employee hierarchy
285 #include <iostream.h>
286 #include <iomanip.h>
```

```

287 #include "employ2.h"
288 #include "boss1.h"
289 #include "commis1.h"
290 #include "piece1.h"
291 #include "hourly1.h"
292
293 void virtualViaPointer( const Employee * );
294 void virtualViaReference( const Employee & );
295
296 int main()
297 {
298     // set output formatting
299     cout << setiosflags( ios::fixed | ios::showpoint )
300         << setprecision( 2 );
301
302     Boss b( "John", "Smith", 800.00 );
303     b.print(); // static binding
304     cout << " earned $" << b.earnings(); // static binding
305     virtualViaPointer( &b ); // uses dynamic binding
306     virtualViaReference( b ); // uses dynamic binding
307
308     CommissionWorker c( "Sue", "Jones", 200.0, 3.0, 150 );
309     c.print(); // static binding
310     cout << " earned $" << c.earnings(); // static binding
311     virtualViaPointer( &c ); // uses dynamic binding
312     virtualViaReference( c ); // uses dynamic binding
313
314     PieceWorker p( "Bob", "Lewis", 2.5, 200 );
315     p.print(); // static binding
316     cout << " earned $" << p.earnings(); // static binding
317     virtualViaPointer( &p ); // uses dynamic binding
318     virtualViaReference( p ); // uses dynamic binding
319
320     HourlyWorker h( "Karen", "Price", 13.75, 40 );
321     h.print(); // static binding
322     cout << " earned $" << h.earnings(); // static binding
323     virtualViaPointer( &h ); // uses dynamic binding
324     virtualViaReference( h ); // uses dynamic binding
325     cout << endl;
326     return 0;
327 }
328
329 // Make virtual function calls off a base-class pointer
330 // using dynamic binding.
331 void virtualViaPointer( const Employee *baseClassPtr )
332 {
333     baseClassPtr->print();
334     cout << " earned $" << baseClassPtr->earnings();
335 }
336
337 // Make virtual function calls off a base-class reference
338 // using dynamic binding.
339 void virtualViaReference( const Employee &baseClassRef )
340 {
341     baseClassRef.print();

```

```

342     cout << " earned $" << baseClassRef.earnings();
343 }

```

输出结果:

```

      Boss: John Smith earned $800.00
      Boss: John Smith earned $800.00
      Boss: John Smith earned $800.00
Commission Worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
      Piece worker: Bob Lewis earned $500.00
      Piece worker: Bob Lewis earned $500.00
      Piece worker: Bob Lewis earned $500.00
      Hourly worker: Karen Price earned $550.00
Hourly worker: Karen Price earned $550.00
      Hourly worker: Karen Price earned $550.00

```

图 10.1 Employee 类层次的多态性

驱动程序 main 函数中的四小段代码是类似的, 因此我们只讨论处理 Boss 对象的第一段代码。第 302 行:

```
Boss b("John", "Smith", 800.00);
```

实例化了类 Boss 的派生类对象 b, 并为构造函数提供了参数 (即姓和名以及固定的周薪)。

第 303 行:

```
b.print(); //static binding
```

用圆点成员选择运算符显式地调用类 Boss 中的成员函数 print。在编译时就可以知道被调用函数的对象类型, 所以它是静态关联。使用该调用是为了和用动态关联调用函数 print 做一比较。

第 304 行:

```
cout << " earned $" << b.earnings(); //static binding
```

用圆点成员选择运算符显式地调用类 Boss 中的成员函数 earnings, 这也是一例静态关联。使用该调用是为了和用动态关联调用函数 earnings 做一比较。

第 305 行:

```
virtualViaPointer( &b ); // uses dynamic binding
```

用派生类对象 b 的地址调用函数 virtualViaPointer (第 331 行)。函数在参数 baseClassPtr 中接收这个地址, 该参数声明为 const Employee \*, 这正是实现多态性所必须要做的。

第 306 行:

```
baseClassPtr->print();
```

调用 baseClassPtr 所指向对象的成员函数 print。由于 print 在基类中被声明为虚函数, 因此系统调用了派生类对象的 print 函数 (仍然是多态性行为)。该函数调用是一例动态关联, 即用基类指针调用虚函数, 以便在执行时才确定调用哪一个函数。

第 307 行:

```
cout << " earned $" << baseClassPtr->earnings();
```

调用 `baseClassPtr` 所指向对象的成员函数 `earnings`。由于 `earnings` 在基类中被声明为虚函数，因此系统调用了派生类对象的 `earnings` 函数，这也是动态关联的一个范例。

第 306 行：

```
virtualViaReference( b ); // uses dynamic binding
```

调用函数 `virtualViaReference` (第 339 行) 演示多态性也可以用基类引用调用虚函数来完成。该函数在参数 `baseClassRef` 中接收对象 `b`，该参数声明为 `const Employee &`。这就是通过引用来影响多态行为。

第 341 行：

```
baseClassRef.print();
```

调用 `baseClassRef` 所引用对象的成员函数 `print`。由于 `print` 在基类中被声明为虚函数，因此系统调用了派生类对象的 `print` 函数。该函数调用是一例动态关联，即用基类引用调用函数，以便在执行时才确定调用哪一个函数。

第 342 行：

```
cout << " earned $" << baseClassRef.earnings();
```

调用 `baseClassRef` 所引用对象的成员函数 `earnings`。由于 `earnings` 在基类中被声明为虚函数，因此系统调用了派生类对象的 `earnings` 函数，这也是动态关联的一个范例。

## 10.7 新类和动态关联

对于预先知道所有可能的类的系统来说，多态性和虚函数当然会运行得很好，但是当向系统中添加各种各样的新类时，它们同样也会运行得很好。动态关联（也叫滞后关联）允许向系统中添加新类。对于要被编译的虚函数调用来说，编译时可以不必要知道对象的类型。在运行时，虚函数调用和被调用对象的成员函数相匹配。

屏幕管理程序可以不经重新编译就可以处理添加到系统中的新的显示对象，`draw` 函数的调用还是和原来的一样，新对象自己包含了实际的显示能力，这样就可以很容易地给系统增加功能，同时也鼓励软件复用。

动态关联可以使独立软件供应商（ISV）在不透露其秘密的情况下发行软件。发行的软件可以只包括头文件和对象文件，不必透露源代码。软件开发者可以利用继承机制从 ISV 提供的类中派生出新类。和 ISV 提供的类一起运行的软件也能够和派生类一起运行，并且能够通过动态关联使用这些派生类中重定义的虚函数。

10.9 节将介绍综合的多态性实例研究。10.10 节将深入介绍多态、虚函数与动态关联如何在 C++ 中实现。

## 10.8 虚析构函数

用多态性处理动态分配的类层次结构中的对象时存在一个问题。如果 `delete` 运算符用于指向派生类对象的基类指针，而程序中又显式地用该运算符删除每一个对象，那么，不管基类指针所指向的对象是何种类型，也不管每个类的析构函数名是不相同的这样一种情况，系统都会为这些对象调用基类的析构函数。

这种问题有一种简单的解决办法,即将基类析构函数声明为虚析构函数。这样就会使所有派生类的析构函数自动成为虚析构函数(即使它们与基类析构函数名不同)。这时,如果像上面那样使用 delete 运算符时,系统会调用相应类的析构函数。记住,删除派生类对象时,同时删除派生类对象的基类部分,基类析构函数在派生类析构函数之后自动执行。

#### 编程技巧 10.2

如果一个类拥有虚函数,即使该类不需要虚析构函数也给它提供一个虚析构函数,这样能够使该类的派生类包含正确调用的析构函数。

#### 常见编程错误 10.2

将构造函数声明为虚函数是语法错误。构造函数不能是虚函数。

## 10.9 实例研究: 继承接口和实现

下面的范例(见图 10.2)要重新考察上一章中的 Point、Circle、Cylinder 类的层次结构,只不过这里类的层次结构的顶层是抽象基类 Shape。类 Shape 中有一个纯虚函数 printShapeName 和 print,所以它是一个抽象基类。类 Shape 中还包含其他两个虚函数 area 和 volume,它们都有默认的实现(返回 0 值)。类 Point 从类 Shape 中继承了这两个函数的实现,由于点的面积和体积是 0,所以这种继承是合理的。类 Circle 从类 Point 中继承了函数 volume,但 Circle 本身提供了函数 area 的实现。Cylinder 对函数 area 和 volume 提供了自己的实现。

注意,尽管 Shape 是一个抽象基类,但是仍然可以包含某些成员函数的实现,并且这些实现是可继承的。类 Shape 以四个虚函数的形式提供了一个可继承的接口(类层次结构中的所有成员都将包含这些虚函数),该类还提供了要在类层次结构头几层的派生类中使用的一些实现。

#### 软件工程视点 10.8

一个类可以从基类继承接口和(或)实现。为实现继承而设计的层次结构倾向于在高层具有某些功能,为接口继承而设计的层次结构则倾向于在较低层具有某些功能。对于前者,每个新派生类继承基类中定义的一个或几个成员函数,新的派生类使用基类定义;对于后者,基类指定一个或几个函数,层次中每个对象都要一样调用(即有相同的签名),但各个派生类提供自己对该函数的实现方法。

```
1 // Fig. 10.2: shape.h
2 // Definition of abstract base class Shape
3 #ifndef SHAPE_H
4 #define SHAPE_H
5 #include <iostream.h>
6
7 class Shape {
8 public:
9     virtual double area() const { return 0.0; }
10    virtual double volume() const { return 0.0; }
11
12    // pure virtual functions overridden in derived classes
13    virtual void printShapeName() const = 0;
14    virtual void print() const = 0;
15 };
16
17 #endif
18 // Fig. 10.2: point1.h
```

---

```

19 // Definition of class Point
20 #ifndef POINT1_H
21 #define POINT1_H
22 #include "shape.h"
23
24 class Point : public Shape {
25 public:
26     Point( int = 0, int = 0 ); // default constructor
27     void setPoint( int, int );
28     int getX() const { return x; }
29     int getY() const { return y; }
30     virtual void printShapeName() const { cout << "Point: "; }
31     virtual void print() const;
32 private:
33     int x, y; // x and y coordinates of Point
34 };
35
36 #endif
37 // Fig. 10.2: point1.cpp
38 // Member function definitions for class Point
39 #include "point1.h"
40
41 Point::Point( int a, int b ) { setPoint( a, b ); }
42
43 void Point::setPoint( int a, int b )
44 {
45     x = a;
46     y = b;
47 }
48
49 void Point::print() const
50 { cout << '[' << x << ", " << y << ']' ; }
51 // Fig. 10.2: circle1.h
52 // Definition of class Circle
53 #ifndef CIRCLE1_H
54 #define CIRCLE1_H
55 #include "point1.h"
56
57 class Circle : public Point {
58 public:
59     // default constructor
60     Circle( double r = 0.0, int x = 0, int y = 0 );
61
62     void setRadius( double );
63     double getRadius() const;
64     virtual double area() const;
65     virtual void printShapeName() const { cout << "Circle: "; }
66     virtual void print() const;
67 private:
68     double radius; // radius of Circle
69 };
70
71 #endif
72 // Fig. 10.2: circle1.cpp
73 // Member function definitions for class Circle
74 #include "circle1.h"

```

```
75
76 Circle::Circle( double r, int a, int b )
77     : Point( a, b ) // call base-class constructor
78 { setRadius( r ); }
79
80 void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }
81
82 double Circle::getRadius() const { return radius; }
83
84 double Circle::area() const
85     { return 3.14159 * radius * radius; }
86
87 void Circle::print() const
88 {
89     Point::print();
90     cout << " Radius = " << radius;
91 }
92 // Fig. 10.2: cylindr1.h
93 // Definition of class Cylinder
94 #ifndef CYLINDR1_H
95 #define CYLINDR1_H
96 #include "circle1.h"
97
98 class Cylinder : public Circle {
99 public:
100     // default constructor
101     Cylinder( double h = 0.0, double r = 0.0,
102             int x = 0, int y = 0 );
103
104     void setHeight( double );
105     double getHeight() const;
106     virtual double area() const;
107     virtual double volume() const;
108     virtual void printShapeName() const { cout << "Cylinder: "; }
109     virtual void print() const;
110 private:
111     double height; // height of Cylinder
112 };
113
114 #endif
115 // Fig. 10.2: cylindr1.cpp
116 // Member and friend function definitions for class Cylinder
117 #include "cylindr1.h"
118
119 Cylinder::Cylinder( double h, double r, int x, int y )
120     : Circle( r, x, y ) // call base-class constructor
121 { setHeight( h ); }
122
123 void Cylinder::setHeight( double h )
124     { height = h > 0 ? h : 0; }
125
126 double Cylinder::getHeight() const { return height; }
127
128 double Cylinder::area() const
129 {
130     // surface area of Cylinder
```

```

131     return 2 * Circle::area() +
132           2 * 3.14159 * getRadius() * height;
133 }
134
135 double Cylinder::volume() const
136 { return Circle::area() * height; }
137
138 void Cylinder::print() const
139 {
140     Circle::print();
141     cout << "; Height = " << height;
142 }
143 // Fig. 10.2: fig10_02.cpp
144 // Driver for shape, point, circle, cylinder hierarchy
145 #include <iostream.h>
146 #include <iomanip.h>
147 #include "shape.h"
148 #include "point1.h"
149 #include "circle1.h"
150 #include "cylindrl.h"
151
152 void virtualViaPointer( const Shape * );
153 void virtualViaReference( const Shape & );
154
155 int main()
156 {
157     cout << setiosflags( ios::fixed | ios::showpoint )
158           << setprecision( 2 );
159
160     Point point( 7, 11 );           // create a Point
161     Circle circle( 3.5, 22, 8 );    // create a Circle
162     Cylinder cylinder( 10, 3.3, 10, 10 ); // create a Cylinder
163
164     point.printShapeName();         // static binding
165     point.print();                  // static binding
166     cout << '\n';
167
168     circle.printShapeName();        // static binding
169     circle.print();                 // static binding
170     cout << '\n';
171
172     cylinder.printShapeName();      // static binding
173     cylinder.print();              // static binding
174     cout << "\n\n";
175
176     Shape *arrayOfShapes[ 3 ];     // array of base-class pointers
177
178     // aim arrayOfShapes[ 0 ] at derived-class Point object
179     arrayOfShapes[ 0 ] = &point;
180
181     // aim arrayOfShapes[ 1 ] at derived-class Circle object
182     arrayOfShapes[ 1 ] = &circle;
183
184     // aim arrayOfShapes[ 2 ] at derived-class Cylinder object
185     arrayOfShapes[ 2 ] = &cylinder;

```



```

186
187 // Loop through arrayOfShapes and call virtualViaPointer
188 // to print the shape name, attributes, area, and volume
189 // of each object using dynamic binding.
190 cout << "Virtual function calls made off "
191      << "base-class pointers\n";
192
193 for ( int i = 0; i < 3; i++ )
194     virtualViaPointer( arrayOfShapes[ i ] );
195
196 // Loop through arrayOfShapes and call virtualViaReference
197 // to print the shape name, attributes, area, and volume
198 // of each object using dynamic binding.
199 cout << "Virtual function calls made off "
200      << "base-class references\n";
201
202 for ( int j = 0; j < 3; j++ )
203     virtualViaReference( *arrayOfShapes[ j ] );
204
205 return 0;
206 }
207
208 // Make virtual function calls off a base-class pointer
209 // using dynamic binding.
210 void virtualViaPointer( const Shape *baseClassPtr )
211 {
212     baseClassPtr->printShapeName();
213     baseClassPtr->print();
214     cout << "\nArea = " << baseClassPtr->area()
215          << "\nVolume = " << baseClassPtr->volume() << "\n\n";
216 }
217
218 // Make virtual function calls off a base-class reference
219 // using dynamic binding.
220 void virtualViaReference( const Shape &baseClassRef )
221 {
222     baseClassRef.printShapeName();
223     baseClassRef.print();
224     cout << "\nArea = " << baseClassRef.area()
225          << "\nVolume = " << baseClassRef.volume() << "\n\n";
226 }

```

**输出结果:**

```

Point: [ 7, 11]
Circle: [ 22, 8]; Radius = 3.50
Cylinder: [ 10, 10]; Radius = 3.30; Height = 10.00

Virtual function calls made off base-class pointers
Point: [ 7, 11]
Area = 0.00
Volume = 0.00

Circle: [ 22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

```

```
Cylinder: [ 10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12

Virtual function calls made off base-class pointers
Point: [ 7, 11]
Area = 0.00
Volume = 0.00

Circle: [ 22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [ 10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
```

图 10.2 定义抽象基类 Shape

基类 Shape 由三个 public 虚函数组成, 不包含任何数据。函数 print 和 printShapeName 是纯虚函数, 因此它们要在每个派生类中重新定义。函数 area 和 volume 都返回 0.0, 当派生类需要对面积 (area) 和 (或) 体积 (volume) 有不同的计算方法时, 这些函数就需要在派生类中重新定义。注意 Shape 是个抽象类, 包含一些“不纯”的虚函数 (area 和 volume)。抽象类可以包含非虚函数和通过派生类继承的数据。

类 Point 是通过 public 继承从类 Shape 派生来的。因为 Point 没有面积和体积 (均为 0.0), 所以类中没有重新定义基类成员函数 area 和 volume, 而是从类 Shape 中继承这两个函数。函数 printShapeName 和 print 是虚函数 (在基类被定义为纯虚函数) 的实现, 如果不在类 Point 中重新定义这些函数, 那么 Point 仍然为抽象类则不能实例化 Point 对象。其他成员函数包括: 将新的 x 和 y 坐标值赋给 Point 对象 (即点) 的一个 “set” 函数和返回 Point 对象的 x 和 y 坐标值的 “get” 函数。

类 Circle 是通过 public 继承从类 Point 派生来的。因为它没有体积, 所以类中没有重新定义基类成员函数 volume, 而是从类 Shape 中继承。Circle 是有面积的, 因此要重新定义函数 area。函数 printShapeName 和 print 是虚函数 (在基类中被定义为纯虚函数) 的实现。如果此处不重新定义该函数, 则会继承类 Point 中该函数的版本。其他成员函数包括为 Circle 对象设置新的 radius (半径值) 的 “set” 函数和返回 Circle 对象的 radius 的 “get” 函数。

类 Cylinder 是通过 public 继承从类 Circle 派生来的。因为 Cylinder 对象的面积和体积同 Circle 的不同, 所以需要在类中重新定义函数 area 和 volume。函数 printShapeName 和 print 是虚函数 (在基类中被定义为纯虚函数) 的实现。如果此处不重新定义该函数, 则会继承类 Circle 中该函数的版本。类中还包括一个设置 Cylinder 对象 height (高度) 的 “set” 函数和一个读取 Cylinder 对象 (圆柱体) 的 height 的 “get” 函数。

驱动程序一开始就分别实例化了类 Point 的对象 point、类 Circle 的对象 circle 和类 Cylinder 的对象 cylinder。程序随后调用了每个对象的 printShapeName 和 print 函数, 并输出每一个对象的信息以验证对象初始化的正确性。每次调用 printShapeName 和 print (第 164 行到第 173 行) 都使用静态关联, 编译器在编译时知道调用 printShapeName 和 print 的每种对象类型。

接着把指针数组 arrayOfShapes 的每个元素声明为 Shape \* 类型, 该数组用来指向每个派生类对象。首先把对象 point 的地址赋给了 arrayOfShapes[ 0 ] (第 179 行)、把对象 circle 的地址赋给了 arrayOfShapes[ 1 ] (第 182 行)、把对象 cylinder 的地址赋给了 arrayOfShapes[ 2 ] (第 185 行)。

然后用for结构(第193行)遍历arrayOfShapes数组,并对每个数组元素调用函数virtualViaPointer(第194行):

```
virtualViaPointer( arrayOfShapes[ i ] );
```

函数 virtualViaPointer 用 baseClassPtr (类型为 const Shape \*) 参数接收 arrayOfShapes 数组中存放的地址。每次执行 virtualViaPointer 时,调用下列4个虚函数:

```
baseClassPtr-> printShapeName()
baseClassPtr-> print()
baseClassPtr-> area()
baseClassPtr-> volume()
```

这些调用方法对执行时 baseClassPtr 所指的对象调用一个虚函数,对象类型无法在编译时确定。输出中显示了对每个类调用的相应函数。首先,输出字符串“Point:”和相应的 point 对象,面积和体积的计算结果都是0.00。然后,输出字符串“Circle:”和 circle 对象的圆心及半径,程序计算出了对象 circle 的面积,返回体积值为0.00。最后,输出字符串“Cylinder:”以及相应的 cylinder 对象的底面圆心、半径和高,程序计算出了对象 cylinder 的面积和体积。所有调用函数 printShapeName、print、area 以及 volume 的虚函数都是在运行时用动态关联解决的。

最后用for结构(第202行)遍历arrayOfShapes数组,并对每个数组元素调用函数virtualViaReference(第203行):

```
virtualViaReference( * arrayOfShapes[ j ] );
```

函数 virtualViaReference 用 baseClassRef (类型为 const Shape &) 参数接收对 arrayOfShapes 数组中存放的地址的引用(通过复引用)。每次执行 virtualViaReference 时,调用下列4个虚函数:

```
baseClassRef.printShapeName()
baseClassRef.print()
baseClassRef.area()
baseClassRef.volume()
```

这些调用方法对执行时 baseClassRef 所指的对象调用上述函数。输出中使用基类引用与使用基类指针时产生的结果是相同的。

## 10.10 多态、虚函数和动态关联

C++中的多态比较容易编程。虽然还可以像C语言等非面向对象语言中一样进行多态编程,但这种做法既复杂,又危险,需要进行指针操作。本节介绍 C++ 如何在内部实现多态、虚函数和动态关联,以便了解这些功能是如何实现的,更重要的是帮助读者了解多态的开销(除了内存占用和处理器时间)。这样就可以更清楚地确定何时使用多态,何时不用多态。第20章“标准模板库(STL)”中将介绍 STL 组件不用多态和虚函数,从而避免运行开销,达到符合 STL 特定要求的最优性能。

首先,我们要介绍 C++ 编译器在编译时建立怎样的数据结构来支持运行时的多态。然后我们介绍执行程序如何利用这些数据结构执行虚函数和实现与多态相关的动态关联。

C++ 编译有一个或几个虚函数的类时,对该类建立虚函数表(virtual function table, vtable)。vtable 让执行程序选择每次执行类的虚函数时正确的实现方法。图10.3演示了 Shape、Point、Circle 和 Cylinder 类的虚函数表。

Shape 类的 vtable 中, 第一个指针指向该类 area 函数的实现方法, 即返回面积 0.0 的函数。第二个指针指向该类 volume 函数的实现方法, 即返回体积 0.0 的函数。printShapeName 和 print 函数都是纯虚函数, 没有实现方法, 因此函数指针都设置为 0。类中的 vtable 有一个或几个 0 指针时, 称为抽象类。类中的 vtable 没有 0 指针时, 称为具体类 (如 Point、Circle 和 Cylinder)。

Point 类继承 Shape 类的 area 和 volume 函数, 因此编译器只是把 Point 类 vtable 表中的这两个指针设为 Shape 类中 area 和 volume 指针的副本。Point 类将函数 printShapeName 重定义为打印 "Point:", 使函数指针指向 Point 类的 printShapeName 函数。Point 类还重定义 print, 使相应函数指针指向 Point 类打印 [x, y] 的函数。

Circle 类 vtable 表中 Circle 的 area 函数指针指向 Circle 的 area 函数 (返回  $\pi r^2$ )。volume 函数指针只是从 Point 类复制, 是原先从 Shape 向 Point 复制的指针。printShapeName 函数指针指向 Circle 版本打印 "Circle:" 的函数。print 函数指针指向 Circle 类的打印 [x, y] 的函数。

Cylinder 类 vtable 表中 Cylinder 的 area 函数指针指向 Cylinder 的 area 函数, 该函数计算 Cylinder 的表面积  $2\pi r^2 + 2\pi rh$ 。Cylinder 的 volume 函数指针指向 volume 函数, 返回  $\pi r^2 h$ 。Cylinder 的 printShapeName 函数指针指向打印 "Cylinder:" 的函数。Cylinder 的 print 函数指针指向 Cylinder 类打印 [x, y] 的函数。

多态是通过复杂的数据结构实现的, 涉及三层指针。前面只介绍了其中一层, 即 vtable 中的函数指针。这些指针在调用虚函数时指向实际执行的函数。

下面要考虑第二层指针。实例化带虚函数的类对象时, 编译器在对象前面连接该类的 vtable 指针 (注意: 这个指针通常放在对象前面, 但也不一定非要这样实现)。

第三层指针是接受虚函数调用的对象句柄 (这个句柄也可以是个引用)。

下面看看典型的虚函数调用如何执行。考虑函数 virtualViaPointer 中的下列调用:

```
baseClassPtr-> printShapeName()
```

假设 baseClassPtr 包含 arrayOfShapes[1] 的指针, 即对象 circle 的地址。则编译器编译这条语句时, 它确定调用实际上是对基类指针进行, 并且 printShapeName 是个虚函数。

然后编译器确定 printShapeName 是每个 vtable 表中的第三个项目。要找到这个项目, 编译器发现需要跳过前两个项目。为此, 编译器编译 8 个字节的偏移量或位移量 (在目前流行的 32 位机器中, 每个指针为 4 个字节), 将其编译到机器语言目标码中, 用于执行虚函数调用。

然后编译器产生完成下列工作的代码 (说明: 下列编号对应图 10.3 中圆圈内的数字):

1. 从 arrayOfShapes 中选择第 i 个项目 (这里是对象 circle 的地址) 并将其传递给 virtualViaPointer, 从而将 baseClassPtr 设置为指向 circle。
2. 复引用指针, 取得 circle 对象, 它以指向 Circle vtable 的指针开始。
3. 复引用 circle 的 vtable 指针, 取得 Circle vtable。
4. 跳过 8 个字节位移, 选择 printShapeName 函数指针。
5. 复引用 printShapeName 函数指针, 构成要执行的实际函数名, 并用函数调用运算符() 执行相应的 printShapeName 函数和打印字符串 "Circle:"。

图 10.3 的数据结构看起来有点复杂, 但这些细节大部分由编译器负责, 程序员不必担心, C++ 中的多态编程并不复杂。

每个虚函数调用中发生的指针复引用操作和内存访问需要增加一些执行时间。vtable 和 vtable 指针要占用一些内存。

现在, 已经有了关于虚函数调用如何工作的足够细节, 可以确定其是否适合具体的应用程序。

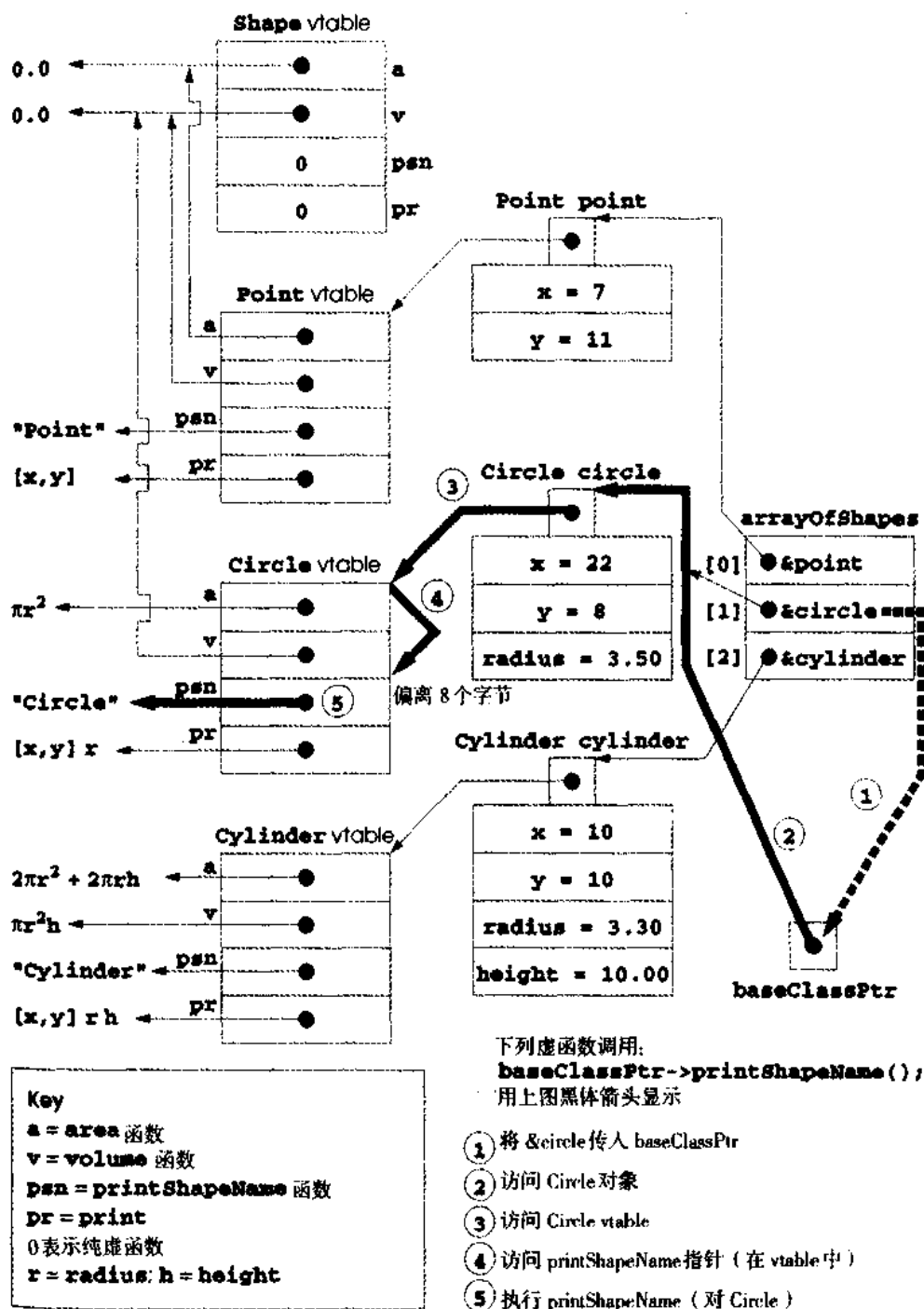


图 10.3 虚函数调用的控制流程图

## 性能提示 10.1

多态性 (它是用虚函数和动态关联实现的) 是高效的, 程序员使用这种功能对系统性能的影响极小。

## 性能提示 10.2

虚函数和动态关联使得多态性编程和 switch 逻辑编程形成了对照。C++ 优化编译器通常能生成至少和手写的基于 switch 逻辑的代码具有同样效率的代码。对大多数应用程序而言, 多态的开销是可以接受的。但有时则不能接受多态的开销, 例如性能要求很高的实时应用程序。

## 小结

- 虚函数和多态性使得设计和实现易于扩展的系统成为可能。在程序开发过程中，不论类是否已经建立，程序员都可以利用虚函数和多态性编写处理这些类对象的程序。
- 虚函数和多态性的程序设计无需使用 switch 逻辑。程序员可以用虚函数机制自动完成等价的逻辑，因而避免与 switch 逻辑有关的各种各样的错误。如果要让客户代码确定对象类型和表达，则是低质的类设计。
- 派生类在需要的时候可以自己提供基类的虚函数实现，否则就使用基类的实现。
- 如果通过用名字和圆点成员选择运算符引用一个特定的对象来调用虚函数，则引用是在编译时确定的（称为静态关联），被调用的虚函数是为该特定对象的类定义的函数或继承该类的函数。
- 在许多情况下，定义不实例化为任何对象的类很有用处，这种类称为“抽象类”。因为抽象类要作为基类被其他类继承，所以通常也把它称为“抽象基类”。抽象基类不能用来建立实例化的对象。
- 可以建立实例化对象的类称为具体类。
- 将带有虚函数的类中的一个或者多个虚函数声明为纯虚函数，则该类就成为抽象类。纯虚函数是在声明时“初始化值”为 0 的函数。
- 如果某个类是从一个带有纯虚函数的类派生出来，并且没有在该派生类中提供该纯虚函数的定义，则该虚函数在派生类中仍然是纯虚函数，因而该派生类也是一个抽象类（不能有任何对象）。
- C++ 支持多态性。所谓多态性是指：通过继承而相关的不同的类，他们的对象能够对同一个函数调用做出不同的响应。
- 多态性是通过虚函数实现的。
- 当通过基类指针或引用请求使用虚函数时，C++ 会在与对象关联的派生类中正确的选择重定义的函数。
- 使用虚函数和多态性能够使成员函数的调用根据接收到该调用的对象的类型产生不同的动作。
- 尽管不能实例化抽象基类的对象，但是可以声明抽象基类的指针。当实例化了具体类的对象后，可以用这种指针使派生类对象具有多态操作能力。
- 使用动态关联（也叫滞后关联）可以向系统中添加新类。对于要被编译的虚函数调用，编译时可以不必要知道对象的类型。在运行时，虚函数调用和被调用对象的成员函数相匹配。
- 动态关联可以使独立软件供应商（ISV）在不透露其秘密的情况下发行软件。发行的软件可以只包括头文件和对象文件，不必透露源代码。软件开发者可以利用继承机制从 ISV 提供的类中派生出新类。和 ISV 提供的类一起运行的软件也能够和派生类一起运行，并且能够使用（通过动态关联）这些派生类中重定义的虚函数。
- 动态关联要求在运行时把对虚函数的调用转换为对应类的虚函数版本。虚函数表（称为 vtable）实现为包含函数指针的数组，每一个包含虚函数的类都有一个 vtable。对于类中的每一个虚函数，vtable 都有一个包含一个函数指针的项目，该指针指向类的对象所要使用的虚函数版本。特定类所要使用的虚函数可能是该类中重新定义的函数，也可能是从较高层的基类直接或间接继承来的函数。

- 当基类提供了一个成员函数并将它声明为 virtual 时, 派生类可以但不是必须重定义该虚函数, 因此派生类可以使用基类的虚函数版本, 这会在 vtable 中指明。
- 带有虚函数的类的每一个对象都包含一个指向该类 vtable 的指针。系统在运行时会获取并复引用正确的函数指针来完成函数调用, 查找 vtable 和复引用指针只需要极少的运行时间的开销, 一般少于最优的客户代码。
- 如果基类中包含虚函数, 把其析构函数声明为虚析构函数。这样做将会使所有派生类的析构函数自动成为虚析构函数(即使它们与基类析构函数的函数名不同)。这时, 如果 delete 运算符用于指向派生类对象的基类指针, 而程序中又显式地用该运算符删除每一个对象, 那么系统会调用相应类的析构函数。
- 任何类在 vtable 中有一个或几个 0 指针时就成为抽象类。而没有 0 指针时就成为具体类(如 Point、Circle 和 Cylinder)。

## 术语

|                                                     |              |                                  |            |
|-----------------------------------------------------|--------------|----------------------------------|------------|
| abstract base class                                 | 抽象基类         | offset into vtable               | vtable 偏移量 |
| abstract class                                      | 抽象类          | override a pure virtual function | 重定义纯虚函数    |
| base-class virtual function                         | 基类虚函数        | override a virtual function      | 重定义虚函数     |
| class hierarchy                                     | 类层次          | pointer to a base class          | 基类指针       |
| concrete class                                      | 具体类          | pointer to a derived class       | 派生类指针      |
| convert derived-class pointer to base-class pointer | 将派生类指针变为基类指针 | pointer to an abstract class     | 抽象类指针      |
| derived class                                       | 派生类          | polymorphism                     | 多态         |
| derived-class constructor                           | 派生类构造函数      | programming "in the general"     | 常规编程       |
| direct base class                                   | 直接基类         | programming "in the specific"    | 特定编程       |
| displacement into vtable                            | vtable 位移    | pure virtual function (=0)       | 纯虚函数(=0)   |
| dynamic binding                                     | 动态关联         | reference to a base class        | 基类引用       |
| early binding                                       | 提前关联         | reference to a derived class     | 派生类引用      |
| eliminating switch statements                       | 消除 switch 语句 | reference to an abstract class   | 抽象类引用      |
| explicit pointer conversion                         | 显式指针转换       | software reusability             | 软件复用性      |
| extensibility                                       | 可扩展性         | static binding                   | 静态关联       |
| implementation inheritance                          | 实现继承         | switch logic                     | switch 逻辑  |
| independent software vendor (ISV)                   | 独立软件供应商      | virtual destructor               | 虚析构函数      |
| indirect base class                                 | 间接基类         | virtual function                 | 虚函数        |
| inheritance                                         | 继承           | virtual function table           | 虚函数表       |
| interface inheritance                               | 接口继承         | vtable                           |            |
| late binding                                        | 滞后关联         | vtable pointer                   | vtable 指针  |

## 自测练习

### 10.1 填空

- 使用继承和多态性有助于消除 \_\_\_\_\_ 逻辑。
- 在类定义中, 将 \_\_\_\_\_ 置于虚函数的函数原型的末尾可以声明该函数为纯虚函数。

- e) 如果一个类包含一个或多个纯虚函数, 则该类为 \_\_\_\_\_。
- d) 在编译时就解决的函数调用称为 \_\_\_\_\_ 关联。
- e) 在运行时才解决的函数调用称为 \_\_\_\_\_ 关联。

## 自测练习答案

10.1 a) switch。b) =0。c) 抽象基类。d) 静态。e) 动态。

## 练习

- 10.2 什么是虚函数? 举一个适合使用虚函数的例子。
- 10.3 构造函数不能是虚函数。怎样使构造函数具有虚函数的效果?
- 10.4 多态如何让程序“一般化”而不是“特殊化”。说明“一般化”编程的主要好处。
- 10.5 说明用 switch 逻辑编程的问题。请解释为什么多态可以代替 switch 逻辑。
- 10.6 区分静态关联与动态关联。请解释动态关联中虚函数和 vtable 的用法。
- 10.7 区分继承接口与继承实现的方法, 继承接口的继承层次设计与继承实现的继承层次设计有什么不同?
- 10.8 区分虚函数与纯虚函数。
- 10.9 (判断对错) 抽象基类中所有虚函数都要声明为纯虚函数。
- 10.10 对本章介绍的 Shape 层次提出一层或几层抽象基类(第一层是 Shape, 第二层包括类 TwoDimensionalShape 和 ThreeDimensionalShape)。
- 10.11 多态如何促进可扩展性?
- 10.12 要求开发一个详细描述图形输出的飞行模拟程序。说明多态对这类问题为什么特别有用。
- 10.13 开发一个基本图形包。用 Shape 类继承层次, 只限于二维形状, 如正方形、长方形、三角形和圆。并与用户交互, 让用户指定每个形状的位置、尺寸、形状和填充字符。用户可以指定多个同一形状的项目。生成每个形状时, 将每个新 Shape 对象的 Shape\* 指针放在数组中。每个类有自己的 draw 成员函数。编写一个多态屏幕管理程序, 遍历数组(可用迭代器)。向数组中的每个对象发一个 draw 消息, 形成屏幕图形。每次用户指定新形状时, 重新输出屏幕图形。
- 10.14 修改图 10.1 的工资系统, 增加 private 数据成员 birthDate (Date 对象) 和 departmentCode (int 类型) 到 Employee 中。假设工资系统每月处理一次。这样, 程序计算每个员工的工资时(多态), 遇到过生日的员工多发 100 美元奖金。
- 10.15 练习 9.14 开发了形状类 Shape 的层次结构, 并在该结构中定义了若干类。修改该层次结构, 使 Shape 成为一个包含接口(供层次结构中的类使用)的抽象基类。从类 Shape 派生出二维形状类 TwoDimensionalShape 和三维形状类 ThreeDimensionalShape, 它们也都是抽象类, 然后用虚函数 print 输出每个类的类型和维数。为了计算类层次结构中每个具体类的对象, 这两个类中还要包括虚函数 area 和 volume。最后再编写一个驱动程序测试类 Shape 的层次结构。



## 第11章 C++ 输入/输出流

### 教学目标

- 了解怎样使用 C++ 面向对象的输入/输出流
- 能够格式化输入和输出
- 了解 I/O 流类的层次结构
- 了解怎样输入/输出用户自定义类型的对象
- 能够建立用户自定义的流操纵算子
- 能够确定输入/输出操作的成功与失败
- 能够把输出流连到输入流上

### 11.1 简介

C++ 标准库提供了一组扩展的输入/输出 (I/O) 功能。本章将详细介绍 C++ 中最常用的一些 I/O 操作, 并对其余的输入/输出功能做一简要的概述。本章的有些内容已经在前面提到, 这里对输入/输出功能做一个更全面的介绍。

本章讨论的许多输入/输出功能都是面向对象的, 读者会发现 C++ 的 I/O 操作能够实现许多功能。C++ 式的 I/O 中还大量利用了 C++ 的其他许多特点, 如引用、函数重载和运算符重载等等。

C++ 使用的是类型安全 (type safe) 的 I/O 操作, 各种 I/O 操作都是以对数据类型敏感的方式来执行的。假定某个函数被专门定义好用来处理某一特定的数据类型, 那么当需要的时候, 该函数会被自动调用以处理所对应的数据类型。如果实际的数据类型和函数之间不匹配, 就会产生编译错误。因此, 错误数据是不可能通过系统检测的 (C 语言则不然。这是 C 语言的漏洞, 会导致某些相当微妙而又奇怪的错误)。

用户既可以指定自定义类型的 I/O, 也可以指定标准类型的 I/O。这种可扩展性是 C++ 最有价值的特点之一。

#### 编程技巧 11.1

尽管 C 语言式的 I/O 在 C++ 中是可以使用的, 但还是应该尽量使用 C++ 特有的 I/O 格式。

#### 软件工程视点 11.1

C++ 式的 I/O 是类型安全的。

#### 软件工程视点 11.2

C++ 允许对预定义类型和自定义类型的 I/O 用同样的方法处理, 从而可加快软件的开发过程, 特别是能够提高软件的复用性。

## 11.2 流

C++ 的 I/O 是以字节流的形式实现的，流实际上就是一个字节序列。在输入操作中，字节从输入设备（如键盘、磁盘、网络连接等）流向内存；在输出操作中，字节从内存流向输出设备（如显示器、打印机、磁盘、网络连接等）。

应用程序把字节的含义与字节关联起来。字节可以是 ASCII 字符、内部格式的原始数据、图形图像、数字音频、数字视频或其他任何应用程序所需要的信息。

输入/输出系统的任务实际上就是以一种稳定、可靠的方式在设备与内存之间传输数据。传输过程中通常包括一些机械运动，如磁盘和磁带的旋转、在键盘上击键等等，这个过程所花费的时间要比处理器处理数据的时间长得多，因此要使性能发挥到最大程度就需要周密地安排 I/O 操作。一些介绍操作系统的书籍（见参考文献 Dc90）深入地讨论了这类问题。

C++ 提供了低级和高级 I/O 功能。低级 I/O 功能（即无格式 I/O）通常只在设备和内存之间传输一些字节。这种传输过程以单个字节为单位，它确实能够提供高速度并且可以大容量的传输，但是使用起来不太方便。

人们通常更愿意使用高级 I/O 功能（即格式化 I/O）。高级 I/O 把若干个字节组合成有意义的单位，如整数、浮点数、字符、字符串以及用户自定义类型的数据。这种面向类型的 I/O 功能适合于大多数情况下的输入输出，但在处理大容量的 I/O 时不是很好。

### 性能提示 11.1

处理大容量文件最好使用无格式 I/O。

### 11.2.1 iostream 类库的头文件

C++ 的 iostream 类库提供了数百种 I/O 功能，iostream 类库的接口部分包含在几个头文件中。

头文件 iostream.h 包含了操作所有输入/输出流所需的基本信息，因此大多数 C++ 程序都应该包含这个头文件。头文件 iostream.h 中含有 cin、cout、cerr、clog 4 个对象，对应于标准输入流、标准输出流、非缓冲和经缓冲的标准错误流。该头文件提供了无格式 I/O 功能和格式化 I/O 功能。

在执行格式化 I/O 时，如果流中带有含参数的流操纵算子，头文件 iomanip.h 所包含的信息是有用的。

头文件 fstream.h 所包含的信息对由用户控制的文件处理操作比较重要。第 13 章将在文件处理程序中使用这个头文件。

每一种 C++ 版本通常还包括其他一些与 I/O 相关的库，这些库提供了特定系统的某些功能，如控制专门用途的音频和视频设备。

### 11.2.2 输入/输出流类和对象

iostream 类库包含了许多用于处理大量 I/O 操作的类。其中，类 istream 支持流输入操作，类 ostream 支持流输出操作，类 iostream 同时支持流输入和输出操作。

类 istream 和类 ostream 是通过单一继承从基类 ios 派生而来的。类 iostream 是通过多重继承从类 istream 和 ostream 派生而来的。继承的层次结构见图 11.1。

运算符重载为完成输入/输出提供了一种方便的途径。重载的左移位运算符（<<）表示流的输出，称为流插入运算符；重载的右移位运算符（>>）表示流的输入，称为流读取运算符。这两个运算符可以和标准流对象 cin、cout、cerr、clog 以及用户自定义的流对象一起使用。



图 11.1 输入 / 输出流类的部分层次结构

`cin` 是类 `istream` 的对象，它与标准输入设备（通常指键盘）连在一起。下面的语句用流读取运算符把整数变量 `grade`（假设 `grade` 为 `int` 类型）的值从 `cin` 输入到内存中。

```
cin >> grade;
```

注意，流读取运算符完全能够识别所处理数据的类型。假设已经正确地声明了 `grade` 的类型，那么没有必要为指明数据类型而给流读取运算符添加类型信息。

`cout` 是类 `ostream` 的对象，它与标准输出设备（通常指显示设备）连在一起。下面的语句用流插入运算符 `cout` 把整型变量 `grade` 的值从内存输出到标准输出设备上。

```
cout << grade;
```

注意，流插入运算符完全能够识别变量 `grade` 的数据类型，假定已经正确地声明了该变量，那么没有必要再为指明数据类型而给流插入运算符添加类型信息。

`cerr` 是类 `ostream` 的对象，它与标准错误输出设备连在一起。到对象 `cerr` 的输出是非缓冲输出，也就是说插入到 `cerr` 中的输出会被立即显示出来，非缓冲输出可迅速把出错信息告诉用户。

`clog` 是类 `ostream` 的对象，它与标准错误输出设备连在一起。到对象 `clog` 的输出是缓冲输出。即每次插入 `clog` 可能使其输出保持在缓冲区，要等缓冲区刷新时才输出。

C++ 中的文件处理用类 `ifstream` 执行文件的输入操作，用类 `ofstream` 执行文件的输出操作，用类 `fstream` 执行文件的输入 / 输出操作。类 `ifstream` 继承了类 `istream`，类 `ofstream` 继承了类 `ostream`，类 `fstream` 继承了类 `iostream`。与 I/O 相关的类的继承关系见图 11.2。虽然多数系统所支持的完整的输入 / 输出流类层次结构中还有很多类，但这里列出的类能够实现多数程序员所需要的绝大部分功能。如果想更多地了解有关文件处理的内容，可参看 C++ 系统中的类库指南。

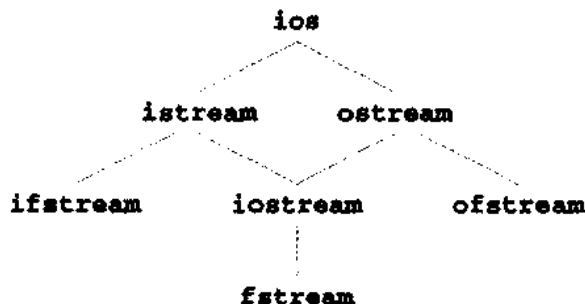


图 11.2 重载的文件处理类的继承层次

## 11.3 输出流

C++ 的类 `ostream` 提供了格式化输出和无格式输出的功能。输出功能包括：用流插入运算符输

出标准类型的数据；用成员函数 `put` 输出字符；成员函数 `write` 的无格式化输出（11.5 节）；输出十进制、八进制、十六进制格式的整数（11.6.1 节）；输出各种精度的浮点数（11.6.2 节）、输出强制带有小数点的浮点数（11.7.2 节）以及用科学计数法和定点计数法表示的浮点数（11.7.6 节）；输出在指定域宽内对齐的数据（11.7.3 节）；输出在域宽内用指定字符填充空位的数据（11.7.4 节）；输出科学计数法和十六进制计数法中的大写字母（11.7.7 节）。

### 11.3.1 流插入运算符

流插入运算符（即重载的运算符 `<<`）可实现流的输出。重载运算符 `<<` 是为了输出内部类型的数据项、字符串和指针值。11.9 节要详细介绍如何用重载运算符 `<<` 输出用户自定义类型的数据项。图 11.3 中的范例程序用一条流插入语句显示了输出的字符串。图 11.4 中的范例程序用多条流插入语句显示输出的字符串，该程序的运行结果与图 11.3 中程序的运行结果相同。

```
1 // Fig. 11.3: fig11_03.cpp
2 // Outputting a string using stream insertion.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome to C++!\n";
8
9     return 0;
10 }
```

**输出结果：**

Welcome to C++!

图 11.3 用一条流插入语句输出字符串

```
1 // Fig. 11.4: fig11_04.cpp
2 // Outputting a string using two stream insertions.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome to ";
8     cout << "C++!\n";
9
10    return 0;
11 }
```

**输出结果：**

Welcome to C++!

图 11.4 用两条流插入语句输出字符串

也可以用流操纵算子 `endl`（行结束）实现转义序列 `\n`（换行符）的功能（见图 11.5）。流操纵算子 `endl` 发送一个换行符并刷新输出缓冲区（不管输出缓冲区是否已满都把输出缓冲区中的内容立即输出）。也可以用下面的语句刷新输出缓冲区：

```
cout << flush;
```

11.6 节要详细讨论流操纵算子。

```
1 // Fig. 11.5: fig11_05.cpp
2 // Using the endl stream manipulator.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome to ";
8     cout << "C++!";
9     cout << endl;    // end line stream manipulator
10
11     return 0;
12 }
```

**输出结果:**

Welcome to C++!

图 11.5 使用流操纵算子 endl

流插入运算符还可以输出表达式的值 (见图 11.6)。

```
1 // Fig. 11.6: fig11_06.cpp
2 // Outputting expression values.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "47 plus 53 is ";
8
9     // parentheses not needed; used for clarity
10    cout << ( 47 + 53 );    // expression
11    cout << endl;
12
13    return 0;
14 }
```

**输出结果:**

47 plus 53 is 100

图 11.6 输出表达式的值

#### 编程技巧 11.2

当输出表达式时, 为防止表达式中的运算符与运算符<<之间存在运算符优先级问题, 用括号将表达式括起来。

### 11.3.2 连续使用流插入 / 流读取运算符

重载的运算符<<和>>都可以在一条语句中连续使用 (见图 11.7)。

```
1 // Fig. 11.7: fig11_07.cpp
2 // Cascading the overloaded << operator.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "47 plus 53 is " << ( 47 + 53 ) << endl;
8
9     return 0;
```

```
10 }
```

**输出结果:**

```
47 plus 53 is 100
```

图 11.7 连续使用重载运算符<<

图中多次使用流插入运算符的语句等同于下面的语句:

```
((cout << "47 plus 53 is ")<<47 + 53)<< endl);
```

之所以可以使用这种写法,是因为重载的运算符<<返回了对其左操作数对象(即cout)的引用,因此最左边括号内的表达式:

```
(cout << "47 plus 53 is ")
```

输出了一个指定的字符串,并返回对cout的引用,因而使中间括号内的表达式解释为:

```
(cout << ( 47 + 53 ) )
```

它输出整数值 100,并返回对cout的引用。于是最右边括号内的表达式解释为:

```
cout << endl
```

它输出一个换行符、刷新cout并返回对cout的引用。最后的返回结果未被使用。

### 11.3.3 输出 char \* 类型的变量

C语言式的I/O必须要提供数据类型信息。C++对此作了改进,能够自动判别数据类型。但是,C++中有时还得使用类型信息。例如,我们知道字符串是char\*类型,假定需要输出其指针的值,即字符串中第一个字符的地址,但是重载运算符<<输出的只是以空(null)字符结尾的char\*类型的字符串,因此使用void\*类型来完成上述需求(需要输出指针变量的地址时都可以使用void\*类型)。图11.8中的程序演示了如何输出char\*类型的字符串及其地址,输出的地址是用十六进制格式表示。在C++中,十六进制数以0x或0X打头,11.6.1节、11.7.4节、11.7.5节和11.7.7节要详细介绍控制数值基数的方法。

```
1 // Fig. 11.8: fig11_08.cpp
2 // Printing the address stored in a char* variable
3 #include <iostream.h>
4
5 int main()
6 {
7     char *string = "test";
8
9     cout << "Value of string is: " << string
10         << "\nValue of static_cast< void *>( string ) is: "
11         << static_cast< void*>( string )<<endl;
12     return 0;
13 }
```

**输出结果:**

```
Value of string is:test
```

```
Value of static_cast <void*>( string )is:0x00416D50
```

图 11.8 输出 char \* 类型变量的地址

### 11.3.4 用成员函数 put 输出字符和 put 函数的连续调用

put 成员函数用于输出一个字符，例如语句：

```
cout.put( 'A' );
```

将字符 A 显示在屏幕上。

也可以像下面那样在一条语句中连续调用 put 函数：

```
cout.put( 'A' ).put( '\n' );
```

该语句在输出字符 A 后输出一个换行符。和 << 一样，上述语句中圆点运算符 (.) 从左向右结合，put 成员函数返回调用 put 的对象的引用。还可以用 ASCII 码值表达式调用 put 函数，语句 cout.put(65) 也输出字符 A。

## 11.4 输入流

下面我们要讨论流的输入，这是用流读取运算符（即重载的运算符 >>）实现的。流读取运算符通常会跳过输入流中的空格、tab 键、换行符等等的空白字符，稍后将介绍如何改变这种行为。当遇到输入流中的文件结束符时，流读取运算符返回 0（false）；否则，流读取运算符返回对调用该运算符的对象的引用。每个输入流都包含一组用于控制流状态（即格式化、出错状态设置等）的状态位。当输入类型有错时，流读取运算符就会设置输入流的 failbit 状态位；如果操作失败则设置 badbit 状态位，后面会介绍如何在 I/O 操作后测试这些状态位。11.7 节和 11.8 节详细讨论了流的状态位。

### 11.4.1 流读取运算符

图 11.9 中的范例程序用 cin 对象和重载的流读取运算符 >> 读取了两个整数。注意流读运算符可以连续使用。

```
1 // Fig. 11.9: fig11_09.cpp
2 // Calculating the sum of two integers input from the keyboard
3 // with the cin object and the stream-extraction operator.
4 #include <iostream.h>
5
6 int main()
7 {
8     int x, y;
9
10    cout << "Enter two integers: ";
11    cin >> x >> y;
12    cout << "Sum of " << x << " and " << y << " is: "
13         << ( x + y ) << endl;
14
15    return 0;
16 }
```

**输出结果：**

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

图 11.9 计算用 cin 和流读取运算符从键盘输入的两个整数值之和

如果运算符>>和<<的优先级相对较高就会出现错误。例如，在图 11.10 的程序中，如果条件表达式没有用括号括起来，程序就得不到正确的编译（读者可以试一下）。

```
1 // Fig. 11.10: fig11_10.cpp
2 // Avoiding a precedence problem between the stream-insertion
3 // operator and the conditional operator.
4 // Need parentheses around the conditional expression.
5 #include <iostream.h>
6
7 int main()
8 {
9     int x, y;
10
11     cout << "Enter two integers: ";
12     cin >> x >> y;
13     cout << x << ( x == y ? " is" : " is not" )
14         << " equal to " << y << endl;
15
16     return 0;
17 }
```

**输出结果：**

```
Enter two integers: 7 5
7 is not equal to 5
```

```
Enter two integers: 8 8
8 is equal to 8
```

图 11.10 避免在流插入运算符和条件运算符之间出现优先级错误

**常见编程错误 11.1**

试图从类 ostream 的对象（或其他只有输出功能的流）中读取数据。

**常见编程错误 11.2**

试图把数据写入类 istream 的对象（或其他只有输入功能的流）。

**常见编程错误 11.3**

在流插入运算符<<或流读取运算符>>的优先级相对较高时没有用圆括号强制实现正确的计算顺序。

我们通常在 while 循环结构的首部用流读取运算符输入一系列值。当遇到文件结束符时，读取运算符返回 0（false）。图 11.11 中的程序用于查找某次考试的最高成绩。假定事先不知道有多少个考试成绩，并且用户会输入表示成绩输入完毕的文件结束符。当用户输入文件结束符时，while 循环结构中的条件（cin>>grade）将变为 0（即 false）。

**可移植性提示 11.1**

提示用户如何从键盘结束输入时，让用户输入文件结束符结束输入，而不是提示输入<ctrl>-d（UNIX 与 Macintosh 所使用的）或<ctrl>-z（PC 与 VAX 所使用的）。

在图 11.11 中，cin>>grade 可以作为条件，因为基类 ios（继承 istream 的类）提供一个重载的强制类型转换运算符，将流变成 void\* 类型的指针。如果读取数值时发生错误或遇到文件结束符，则指针值为 0。编译器能够隐式使用 void\* 类型的强制转换运算符。



```
1 // Fig. 11.11: fig11_11.cpp
2 // Stream-extraction operator returning false on end-of-file.
3 #include <iostream.h>
4
5 int main()
6 {
7     int grade, highestGrade = -1;
8
9     cout << "Enter grade (enter end-of-file to end): ";
10    while ( cin >> grade ) {
11        if ( grade > highestGrade )
12            highestGrade = grade;
13
14        cout << "Enter grade (enter end-of-file to end): ";
15    }
16
17    cout << "\n\nHighest grade is: " << highestGrade << endl;
18    return 0;
19 }
```

**输出结果:**

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^z
Highest grade is: 99
```

图 11.11 流读取运算符在遇到文件结束符时返回 false

### 11.4.2 成员函数 get 和 getline

不带参数的 get 函数从指定的输入流中读取 (输入) 一个字符 (包括空白字符), 并返回该字符作为函数调用的值; 当遇到输入流中的文件结束符时, 该版本的 get 函数返回 EOF。

图 11.12 中的程序把成员函数 eof 和 get 用于输入流 cin, 把成员函数 put 用于输出流 cout。程序首先输出了 cin.eof() 的值, 输出值为 0 (false) 表明没有在 cin 中遇到文件结束符。然后, 程序让用户键入以文件结束符结尾的一行文本 (在 IBM PC 兼容系统中, 文件结束符用 <ctrl>-z 表示; 在 UNIX 和 Macintosh 系统中, 文件结束符用 <ctrl>-d 表示)。程序逐个读取所输入的每个字符, 并调用成员函数 put 将所读取的内容送往输出流 cout。当遇到文件结束符时, while 循环终止, 输出 cin.eof() 的值, 输出值为 1 (true) 表示已接收到了 cin 中的文件结束符。注意, 程序中使用了类 istream 中不带参数的 get 成员函数, 其返回值是所输入的字符。

```
1 // Fig. 11.12: fig11_12.cpp
2 // Using member functions get, put and eof.
3 #include <iostream.h>
4
5 int main()
6 {
7     char c;
8
9     cout << "Before input, cin.eof() is " << cin.eof()
```

```

10         << "\nEnter a sentence followed by end-of-file:\n";
11
12     while ( ( c = cin.get() ) != EOF )
13         cout.put( c );
14
15     cout << "\nEOF in this system is: " << c;
16     cout << "\nAfter input, cin.eof() is " << cin.eof() << endl;
17     return 0;
18 )

```

**输出结果:**

```

Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions^z
Testing the get and put member functions
EOF in this system is: -1
After input cin.eof() is 1

```

图 11.12 使用成员函数 get、put 和 eof

带一个字符型参数的 get 成员函数自动读取输入流中的下一个字符(包括空白字符)。当遇到文件结束符时,该版本的函数返回 0,否则返回对 istream 对象的引用,并用该引用再次调用 get 函数。

带有三个参数的 get 成员函数的参数分别是接收字符的字符数组、字符数组的大小和分隔符(默认值为 '\n')。函数或者在读取比指定的最大字符数少一个字符后结束,或者在遇到分隔符时结束。为使字符数组(被程序用作缓冲区)中的输入字符串能够结束,空字符会被插入到字符数组中。函数不把分隔符放到字符数组中,但是分隔符仍然会保留在输入流中。图 11.13 的程序比较了 cin(与流读取运算符一起使用)和 cin.get 的输入结果。注意调用 cin.get 时未指定分隔符,因此用默认的 '\n'。

```

1 // Fig. 11.13: fig11_13.cpp
2 // Contrasting input of a string with cin and cin.get.
3 #include <iostream.h>
4
5 int main()
6 {
7     const int SIZE = 80;
8     char buffer1[ SIZE ], buffer2[ SIZE ];
9
10    cout << "Enter a sentence:\n";
11    cin >> buffer1;
12    cout << "\nThe string read with cin was:\n"
13         << buffer1 << "\n\n";
14
15    cin.get( buffer2, SIZE );
16    cout << "The string read with cin.get was:\n"
17         << buffer2 << endl;
18
19    return 0;
20 }

```

**输出结果:**

```
Enter a sentence:
Contrasting string input with cin and cin.get
The string read with cin was:
Contrasting

The string read with cin.get was:
string input with cin and cin.get
```

图 11.13 比较用 cin 和 cin.get 输入的字符串结果

成员函数 `getline` 与带三个参数的 `get` 函数类似, 它读取一行信息到字符数组中, 然后插入一个空字符。所不同的是, `getline` 要去除输入流中的分隔符 (即读取字符并删除它), 但是不把它存放在字符数组中。图 11.14 中的程序演示了用 `getline` 输入一行文本。

```
1 // Fig. 11.14: fig11_14.cpp
2 // Character input with member function getline.
3 #include <iostream.h>
4
5 int main()
6 {
7     const SIZE = 80;
8     char buffer[ SIZE ];
9
10    cout << "Enter a sentence:\n";
11    cin.getline( buffer, SIZE );
12
13    cout << "\nThe sentence entered is:\n" << buffer << endl;
14    return 0;
15 }
```

**输出结果:**

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

图 11.14 用成员函数 `getline` 输入字符

### 11.4.3 istream 类中的其他成员函数 ( `peek`、`putback` 和 `ignore` )

成员函数 `ignore` 用于在需要时跳过流中指定数量的字符 (默认个数是 1), 或在遇到指定的分隔符 (默认分隔符是 EOF, 使得 `ignore` 在读文件的时候跳过文件末尾) 时结束。

成员函数 `putback` 将最后一次用 `get` 从输入流中提取的字符放回到输入流中。对于某些应用程序, 该函数是很有用的。例如, 如果应用程序需要扫描输入流以查找用特定字符开始的域, 那么当输入该字符时, 应用程序要把该字符放回到输入流中, 这样才能使该字符包含到要被输入的数据中。

成员函数 `peek` 返回输入流中的下一个字符, 但并不将其从输入流中删除。

#### 11.4.4 类型安全的 I/O

C++ 提供类型安全的 I/O。重载运算符<<和>>是为了接收指定类型的数据。如果接收了非法的数据类型,那么系统就会设置各种错误标志,用户可通过测试这些标志判断 I/O 操作的成功与失败。这种处理方法能够使程序保持控制权。11.8 节要讨论这些错误标志。

### 11.5 成员函数 read、gcount 和 write 的无格式输入 / 输出

调用成员函数 read、write 可实现无格式输入/输出。这两个函数分别把一定量的字节写入字符数组和从字符数组中输出。这些字节都是未经任何格式化的,仅仅是以原始数据形式输入或输出。例如:

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write( buffer, 10 );
```

输出 buffer 的 10 个字节 (包括终止 cout 和<<输出的空字符)。因为字符串就是第一个字符的地址,所以函数调用:

```
cout.write( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10 );
```

显示了字母表中的前 10 个字母。

成员函数 read 把指定个数的字符输入到字符数组中。如果读取的字符个数少于指定的数目,可以设置标志位 failbit (见 11.8 节)。

成员函数 gcount 统计最后输入的字符个数。

图 11.15 中的程序演示了类 istream 中的成员函数 read 和 gcount, 以及类 ostream 中的成员函数 write。程序首先用函数 read 向字符数组 buffer 中输入 20 个字符 (输入序列比字符数组长), 然后用函数 gcount 统计所输入的字符个数, 最后用函数 write 输出 buffer 中的字符。

```
1 // Fig. 11.15: fig11_15.cpp
2 // Unformatted I/O with read, gcount and write.
3 #include <iostream.h>
4
5 int main()
6 {
7     const int SIZE = 80;
8     char buffer[ SIZE ];
9
10    cout << "Enter a sentence:\n";
11    cin.read( buffer, 20 );
12    cout << "\nThe sentence entered was:\n";
13    cout.write( buffer, cin.gcount() );
14    cout << endl;
15    return 0;
16 }
```

**输入结果:**

```
Enter a sentence:
Using the read, write, and gcount member functions
```

```
The sentence entered was:  
Using the read, write
```

图 11.15 成员函数 read、gcount 和 write 的无格式 I/O

## 11.6 流操纵算子

C++ 提供了大量的用于执行格式化输入/输出的流操纵算子。流操纵算子提供了许多功能,如设置域宽、设置精度、设置和清除格式化标志、设置域填充字符、刷新流、在输出流中插入换行符并刷新该流、在输出流中插入空字符、跳过输入流中的空白字符等等。下面几节要介绍这些特征。

### 11.6.1 整数流的基数: 流操纵算子 dec、oct、hex 和 setbase

整数通常被解释为十进制(基数为 10)整数。如下方法可改变流中整数的基数: 插入流操纵算子 hex 可设置十六进制基数(基数为 16)、插入流操纵算子 oct 可设置八进制基数(基数为 8)、插入流操纵算子 dec 可恢复十进制基数。

也可以用流操纵算子 setbase 来改变基数, 流操纵算子 setbase 带有一个整数参数 10、8 或 16。因为流操纵算子 setbase 是带有参数的, 所以也称之为参数化的流操纵算子。使用 setbase 或其他任何参数化的操纵算子都必须在程序中包含头文件 iomanip.h。如果不明确地改变流的基数, 流的基数是不变的。图 11.16 中的程序示范了流操纵算子 hex、oct、dec 和 setbase 的用法。

```
1 // Fig. 11.16: fig11_16.cpp  
2 // Using hex, oct, dec and setbase stream manipulators.  
3 #include <iostream.h>  
4 #include <iomanip.h>  
5  
6 int main()  
7 {  
8     int n;  
9  
10    cout << "Enter a decimal number: ";  
11    cin >> n;  
12  
13    cout << n << " in hexadecimal is: "  
14        << hex << n << '\n'  
15        << dec << n << " in octal is: "  
16        << oct << n << '\n'  
17        << setbase( 10 ) << n << " in decimal is: "  
18        << n << endl;  
19  
20    return 0;  
21 }
```

**输入结果:**

```
Enter a decimal number: 20  
20 in hexadecimal is: 14  
20 in octal is: 24  
20 in decimal is: 20
```

图 11.16 使用流操纵算子 hex、oct、dec 和 setbase

### 11.6.2 设置浮点数精度 (precision、setprecision)

在 C++ 中可以人为控制浮点数的精度，也就是说可以用流操纵算子 `setprecision` 或成员函数 `precision` 控制小数点后面的位数。设置了精度以后，该精度对之后所有的输出操作都有效，直到下一次设置精度为止。无参数的成员函数 `precision` 返回当前设置的精度。图 11.17 中的程序用成员函数 `precision` 和流操纵算子 `setprecision` 打印出了 2 的平方根表，输出结果的精度从 0 连续变化到 9。

```
1 // Fig. 11.17: fig11_17.cpp
2 // Controlling precision of floating-point values
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <math.h>
6
7 int main()
8 {
9     double root2 = sqrt( 2.0 );
10    int places;
11
12    cout << setiosflags( ios::fixed)
13         << "Square root of 2 with precisions 0-9.\n"
14         << "Precision set by the "
15         << "precision member function:" << endl;
16
17    for ( places = 0; places <= 9; places++ ) {
18        cout.precision( places );
19        cout << root2 << '\n';
20    }
21
22    cout << "\nPrecision set by the "
23         << "setprecision manipulator:\n";
24
25    for ( places = 0; places <= 9; places++ )
26        cout << setprecision( places ) << root2 << '\n';
27
28    return 0;
29 }
```

**输入结果：**

```
Square root of 2 with precisions 0-9.
Precision set by the precision member function:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
Precision set by the setprecision manipulator:
1
1.4
```

```
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

图 11.17 控制浮点数的精度

### 11.6.3 设置域宽 (setw、width)

成员函数 `ios::width` 设置当前的域宽 (即输入输出的字符数) 并返回以前设置的域宽。如果显示数据所需的宽度比设置的域宽小, 空位用填充字符填充。如果显示数据所需的宽度比设置的域宽大, 显示数据并不会被截断, 系统会输出所有位。

#### 常见编程错误 11.4

域宽设置仅对下一行流读取或流插入操作有效, 在一次操作完成之后, 域宽又被置回 0, 也就是输出值按照所需要的宽度来输出。不带参数的 `width` 函数返回当前域宽。认为域宽设置适用于所有输出是个逻辑错误。

#### 常见编程错误 11.5

未对所处理的输出数据提供足够的域宽时, 输出数据将按需要的域宽进行输出, 有可能产生难以阅读的输出结果。

图 11.18 中的程序示范了成员函数 `width` 的输入和输出操作。注意, 输入操作提取字符串的最大宽度比定义的域宽小 1, 这是因为在输入的字符串后面必须加上一个空字符。当遇到非前导空白字符时, 流读取操作就会终止。流操纵算子 `setw` 也可以用来设置域宽。注意: 提示用户输入时, 用户应输入一行文本并按 Enter 键加上文件结束符 ( <ctrl>-z 对于 IBM PC 兼容系统; <ctrl>-d 对于 UNIX 与 Macintosh 系统 )。

```
1 // fig11_18.cpp
2 // Demonstrating the width member function
3 #include <iostream.h>
4
5 int main()
6 {
7     int w = 4;
8     char string[ 10 ];
9
10    cout << "Enter a sentence:\n";
11    cin.width( 5 );
12
13    while ( cin >> string ) {
14        cout.width( w++ );
15        cout << string << endl;
16        cin.width( 5 );
17    }
18
19    return 0;
20 }
```

输入结果:

```
Enter a sentence:
This is a test of the width member function
This
  is
    a
  test
    of
  the
  width
    h
  memb
    er
  func
    tion
```

图 11.18 演示成员函数 width

#### 11.6.4 用户自定义的流操纵算子

用户可以建立自己的流操纵算子。图 11.19 中的程序说明了如何建立和使用新的流操纵算子 bell、ret、tab 和 endLine。用户还可以建立自己的带参数的流操纵算子，这方面内容可参看系统的手册。

```
1 // Fig. 11.19: fig11_19.cpp
2 // Creating and testing user-defined, nonparameterized
3 // stream manipulators.
4 #include <iostream.h>
5
6 // bell manipulator (using escape sequence \a)
7 ostream& bell( ostream& output ) { return output << '\a'; }
8
9 // ret manipulator (using escape sequence \r)
10 ostream& ret( ostream& output ) { return output << '\r'; }
11
12 // tab manipulator (using escape sequence \t)
13 ostream& tab( ostream& output ) { return output << '\t'; }
14
15 // endLine manipulator (using escape sequence \n
16 // and the flush member function)
17 ostream& endLine( ostream& output )
18 {
19     return output << '\n' << flush;
20 }
21
22 int main()
23 {
24     cout << "Testing the tab manipulator:" << endLine
25         << 'a' << tab << 'b' << tab << 'c' << endLine
26         << "Testing the ret and bell manipulators:"
27         << endLine << ".....";
28     cout << bell;
29     cout << ret << "-----" << endLine;
30     return 0;
```



```

31 }

输入结果:
Testing the tab manipulator:
a      b      c
Testing the ret and bell manipulators:
-----

```

图 11.19 建立并测试用户自定义的、无参数的流操作算子

## 11.7 流格式状态

各种格式标志指定了在 I/O 流中执行的格式类型。成员函数 `setf`、`unsetf` 和 `flag` 控制标志的设置。

### 11.7.1 格式状态标志

图 11.20 中的格式状态标志在类 `ios` 中被定义为枚举值，留到几节解释。

虽然成员函数 `flags`、`setf` 和 `unsetf` 可以控制这些标志，但是许多 C++ 程序员更喜欢使用流操纵算子（见 11.7.8 节）。程序员可以用按位或操作（`|`）把各种标志选项结合在一个 `long` 类型的值中（见图 11.23）。调用成员函数 `flags` 并指定要进行或操作的标志选项就可以在流中设置这些标志，并返回一个包含以前标志选项的 `long` 类型的值。返回值通常要保存起来，以使用保存值调用 `flags` 来恢复以前的流选项。

`flags` 函数必须指定一个代表所有标志设置的值。而带有一个参数的函数 `setf` 可指定一个或多个要进行或操作的标志，并把它们与现存的标志设置相“或”，形成新的格式状态。

参数化的流操纵算子 `setiosflags` 与成员函数 `setf` 执行同样的功能。流操纵算子 `resetiosflags` 与成员函数 `unsetf` 执行同样的功能。不论使用哪一种流操纵算子，在程序中都必须包含头文件 `iosmanip.h`。

`skipws` 标志指示 `>>` 跳过输入流中的空白字符。`>>` 的默认行为是跳过空白字符，调用 `unsetf(ios::skipws)` 可改变默认行为。流操纵算子 `ws` 也可用于指定跳过流中的空白字符。

| 格式状态标志                       | 说明                                                          |
|------------------------------|-------------------------------------------------------------|
| <code>ios::skipws</code>     | 跳过输入流中的空白字符                                                 |
| <code>ios::left</code>       | 在域中左对齐输出，必要时在右边显示填充字符                                       |
| <code>ios::right</code>      | 在域中右对齐输出，必要时在左边显示填充字符                                       |
| <code>ios::internal</code>   | 表示数字的符号应在域中左对齐，而数字值应在域中右对齐（即在符号和数值之间填充字符）                   |
| <code>ios::dec</code>        | 指定整数应作为十进制（基数 10）值                                          |
| <code>ios::oct</code>        | 指定整数应作为八进制（基数 8）值                                           |
| <code>ios::hex</code>        | 指定整数应作为十六进制（基数 16）值                                         |
| <code>ios::showbase</code>   | 指定在数值前面输出进制（0 表示八进制，0x 或 0X 表示十六进制）                         |
| <code>ios::showpoint</code>  | 指定浮点数输出时应带小数点。这通常和 <code>ios::fixed</code> 一起使用保证小数点后面有一定位数 |
| <code>ios::uppercase</code>  | 指定表示十六进制的 x 应为大写，表示浮点数科学计数法的 e 应为大写                         |
| <code>ios::showpos</code>    | 指定正数和负数前面分别加上 + 和 - 号                                       |
| <code>ios::scientific</code> | 指定浮点数输出采用科学计数法                                              |
| <code>ios::fixed</code>      | 指定浮点数输出采用定点符号，保证小数点后面有一定位数                                  |

图 11.20 格式状态标志

### 11.7.2 尾数零和十进制小数点 (ios::showpoint)

设置 showpoint 标志是为了强制输出浮点数的小数点和尾数零。若没有设置 showpoint, 浮点数 79.0 将被打印为 79, 否则打印为 79.000000 (或由当前精度指定的尾数零的个数)。图 11.21 中的程序用成员函数 setf 设置 showpoint 标志, 从而控制了浮点数的尾数零和小数点的输出。

```
1 // Fig. 11.21: fig11_21.cpp
2 // Controlling the printing of trailing zeros and decimal
3 // points for floating-point values.
4 #include <iostream.h>
5 #include <iomanip.h>
6 #include <math.h>
7
8 int main()
9 {
10     cout << "Before setting the ios::showpoint flag\n"
11         << "9.9900 prints as: " << 9.9900
12         << "\n9.9000 prints as: " << 9.9000
13         << "\n9.0000 prints as: " << 9.0000
14         << "\n\nAfter setting the ios::showpoint flag\n";
15     cout.setf( ios::showpoint );
16     cout << "9.9900 prints as: " << 9.9900
17         << "\n9.9000 prints as: " << 9.9000
18         << "\n9.0000 prints as: " << 9.0000 << endl;
19     return 0;
20 }
```

#### 输入结果:

```
Before setting the ios::showpoint flag
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After setting the ios::showpoint flag
9.9900 prints as: 9.990000
9.9000 prints as: 9.900000
9.0000 prints as: 9.000000
```

图 11.21 控制浮点数的尾数零和小数点的输出

### 11.7.3 对齐 (ios::left、ios::right、ios::internal)

left 标志可以使输出域左对齐并把填充字符放在输出数据的右边, right 标志可以使输出域右对齐并把填充字符放在输出数据的左边。填充的字符由成员函数 fill 或参数化的流操纵算子 setfill 指定 (见 11.7.4 节)。图 11.22 用流操纵算子 setw、setiosflags、resetiosflags 以及成员函数 setf 和 unsetf 控制整数值在域宽内左对齐和右对齐。

```
1 // Fig. 11.22: fig11_22.cpp
2 // Left-justification and right-justification.
3 #include <iostream.h>
4 #include <iomanip.h>
5
```

```

6 int main()
7 {
8     int x = 12345;
9
10    cout << "Default is right justified:\n"
11          << setw(10) << x << "\n\nUSING MEMBER FUNCTIONS"
12          << "\nUse setf to set ios::left:\n" << setw(10);
13
14    cout.setf( ios::left, ios::adjustfield );
15    cout << x << "\nUse unsetf to restore default:\n";
16    cout.unsetf( ios::left );
17    cout << setw( 10 ) << x
18          << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
19          << "\nUse setiosflags to set ios::left:\n"
20          << setw( 10 ) << setiosflags( ios::left ) << x
21          << "\nUse resetiosflags to restore default:\n"
22          << setw( 10 ) << resetiosflags( ios::left )
23          << x << endl;
24    return 0;
25 }

```

**输入结果:**

```

Default is right justified:
      12345
USING MEMBER FUNCTIONS
Use setf to set ios::left:
      12345
Use unsetf to restore default:
12345
USING PARAMETERIZED STREAM MANIPULATORS
Use setiosflags to set ios::left:
12345
Use resetiosflags to restore default:
12345

```

图 11.22 左对齐和右对齐

`internal` 标志指示将一个数的符号位 ( 设置了 `ios::showbase` 标志时为基数, 见 11.7.5 节 ) 在域宽内左对齐, 数值右对齐, 中间的空白由填充字符填充。 `left`、`right` 和 `internal` 标志包含在静态数据成员 `ios::adjustfield` 中。在设置 `left`、`right` 和 `internal` 对齐标志时, `setf` 的第二个参数必须是 `ios::adjustfield`, 这样才能使 `setf` 只设置其中一个标志 ( 这三个标志是互斥的 )。图 11.23 中的程序用流操纵算子 `setiosflags` 和 `setw` 指定中间的空白。注意, `ios::showpos` 标志强制打印了加号。

```

1 // Fig. 11.23: fig11_23.cpp
2 // Printing an integer with internal spacing and
3 // forcing the plus sign.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 int main()
8 {
9     cout << setiosflags( ios::internal | ios::showpos )
10          << setw( 10 ) << 123 << endl;

```

```

11     return 0;
12 }

```

**输入结果:**

```

+      123

```

图 11.23 打印中间带空白的整数并强制输出加号

#### 11.7.4 设置填充字符 (fill、setfill)

成员函数 fill 设置用于使输出域对齐的填充字符, 如果不特别指定, 空格即为填充字符。fill 函数返回以前设置的填充字符。图 11.24 中的程序演示了使用成员函数 fill 和流操纵算子 setfill 来控制填充字符的设置和清除。

```

1 // Fig. 11.24: fig11_24.cpp
2 // Using the fill member function and the setfill
3 // manipulator to change the padding character for
4 // fields larger than the values being printed.
5 #include <iostream.h>
6 #include <iomanip.h>
7
8 int main()
9 {
10     int x = 10000;
11
12     cout << x << " printed as int right and left justified\n"
13         << "and as hex with internal justification.\n"
14         << "Using the default pad character (space):\n";
15     cout.setf( ios::showbase );
16     cout << setw( 10 ) << x << '\n';
17     cout.setf( ios::left, ios::adjustfield );
18     cout << setw( 10 ) << x << '\n';
19     cout.setf( ios::internal, ios::adjustfield );
20     cout << setw( 10 ) << hex << x;
21
22     cout << "\n\nUsing various padding characters:\n";
23     cout.setf( ios::right, ios::adjustfield );
24     cout.fill( '*' );
25     cout << setw( 10 ) << dec << x << '\n';
26     cout.setf( ios::left, ios::adjustfield );
27     cout << setw( 10 ) << setfill( '%' ) << x << '\n';
28     cout.setf( ios::internal, ios::adjustfield );
29     cout << setw( 10 ) << setfill( '^' ) << hex << x << endl;
30     return 0;
31 }

```

**输入结果:**

```

10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
      10000
10000

```

```

0x    2710

Using various padding characters:
*****10000
10000%$$$%
0x^^^^2710

```

图 11.24 用 fill 和 setfill 为实际宽度小于域宽的数据改变填充字符

### 11.7.5 整数流的基数: (ios::dec、ios::oct、ios::hex、ios::showbase)

静态成员 ios::basefield (在 setf 中的用法与 ios::adjustfield 类似) 包括 ios::oct、ios::hex 和 ios::dec 标志位, 这些标志位分别指定把整数作为八进制、十六进制和十进制值处理。如果没有设置这些位, 则流插入运算默认整数为十进制数, 流读取运算按整数提供的方式处理数据 (即以零打头的整数按八进制数处理, 以 0x 或 0X 打头的按十六进制数处理, 其他所有整数都按十进制数处理)。一旦为流指定了特定的基数, 流中的所有整数都按该基数处理, 直到指定了新的基数或程序结束为止。

设置 showbase 标志可强制输出整数值的基数。十进制数以通常方式输出, 输出的八进制数以 0 打头, 输出的十六进制数以 0x 或 0X 打头 (由 uppercase 标志决定是 0x 还是 0X, 见 11.7.7 节)。图 11.25 中的程序用 showbase 标志强制整数按十进制、八进制和十六进制格式打印。

```

1 // Fig. 11.25: fig11_25.cpp
2 // Using the ios::showbase flag
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int x = 100;
9
10    cout << setiosflags( ios::showbase )
11         << "Printing integers preceded by their base:\n"
12         << x << '\n'
13         << oct << x << '\n'
14         << hex << x << endl;
15    return 0;
16 }

```

**输入结果:**

```

Printing integers preceded by their base:
100
0144
0x64

```

图 11.25 使用 ios::showbase 标志

### 11.7.6 浮点数和科学记数法 (ios::scientific、ios::fixed)

ios::scientific 和 ios::fixed 标志包含在静态数据成员 ios::floatfield 中 (在 setf 中的用法与 ios::adjustfield 和 ios::basefield 类似)。这些标志用于控制浮点数的输出格式。设置 scientific 标志使浮点

数按科学记数法输出，设置 `fixed` 标志使浮点数按照定点格式输出，即显示出小数点，小数点后边有指定的位数（由成员函数 `precision` 指定）。若没有这些设置，则浮点数的值决定输出格式。

调用 `cout.setf(0, ios::floatfield)` 恢复输出浮点数的系统默认格式。图 11.26 中的程序示范了以定点格式和科学记数法显示的浮点数。

```

1 // Fig. 11.26: fig11_26.cpp
2 // Displaying floating-point values in system default,
3 // scientific, and fixed formats.
4 #include <iostream.h>
5
6 int main()
7 {
8     double x = .001234567, y = 1.946e9;
9
10    cout << "Displayed in default format:\n"
11         << x << '\t' << y << '\n';
12    cout.setf( ios::scientific, ios::floatfield );
13    cout << "Displayed in scientific format:\n"
14         << x << '\t' << y << '\n';
15    cout.unsetf( ios::scientific );
16    cout << "Displayed in default format after unsetf:\n"
17         << x << '\t' << y << '\n';
18    cout.setf( ios::fixed, ios::floatfield );
19    cout << "Displayed in fixed format:\n"
20         << x << '\t' << y << endl;
21    return 0;
22 }

```

#### 输入结果：

```

Displayed in default format:
0.00123457      1.946e+009
Displayed in scientific format:
1.234567e-003   1.946000e+009
Displayed in default format after unsetf:
0.00123457      1.946e+009
Displayed in fixed format:
0.001235        1946000000.000000

```

图 11.26 以系统默认格式、科学计数法和定点格式显示浮点数

### 11.7.7 大/小写控制 ( `ios::uppercase` )

设置 `ios::uppercase` 标志用来使大写的 X 和 E 分别同十六进制整数和以科学记数法表示的浮点数一起输出（见图 11.27）一旦设置 `ios::uppercase` 标志，十六进制数中的所有字母都转换成大写。

```

1 // Fig. 11.27: fig11_27.cpp
2 // Using the ios::uppercase flag
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     cout << setiosflags( ios::uppercase )

```

```

9         << "Printing uppercase letters in scientific\n"
10        << "notation exponents and hexadecimal values:\n"
11        << 4.345e10 << '\n' << hex << 123456789 << endl;
12    return 0;
13}

```

**输入结果:**

```

Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+010
75BCD15

```

图 11.27 使用 ios::uppercase 标志

**11.7.8 设置及清除格式标志 ( flags、setiosflags、resetiosflags )**

无参数的成员函数 flags 只返回格式标志的当前设置 ( long 类型的值 )。带一个 long 类型参数的成员函数 flags 按参数指定的格式设置标志, 返回以前的标志设置。flags 的参数中未指定的任何格式都会被清除。图 11.28 的程序用成员函数 flags 设置新的格式状态和保存以前的格式状态, 然后恢复原来的格式设置。

```

1 // Fig. 11.28: fig11_28.cpp
2 // Demonstrating the flags member function.
3 #include <iostream.h>
4
5 int main()
6 {
7     int i = 1000;
8     double d = 0.0947628;
9
10    cout << "The value of the flags variable is: "
11        << cout.flags()
12        << "\nPrint int and double in original format:\n"
13        << i << '\t' << d << "\n\n";
14    long originalFormat =
15        cout.flags( ios::oct | ios::scientific );
16    cout << "The value of the flags variable is: "
17        << cout.flags()
18        << "\nPrint int and double in a new format\n"
19        << "specified using the flags member function:\n"
20        << i << '\t' << d << "\n\n";
21    cout.flags( originalFormat );
22    cout << "The value of the flags variable is: "
23        << cout.flags()
24        << "\nPrint values in original format again:\n"
25        << i << '\t' << d << endl;
26    return 0;
27 }

```

**输入结果:**

```

The value of the flags variable is: 0
Print int and double in original format:
1000      0.0947628

```

```
The value of the flags variable is: 4040
Print int and double in a new format
Specified using the flags member function:
1750      9.476280e-002

The value of the flags variable is: 0
Print values in original format again:
1000      0.0947628
```

图 11.28 演示成员函数 flags

成员函数 `setf` 设置参数所指定的格式标志, 并返回 `long` 类型的标志设置值, 例如:

```
long previousFlagSettings=
    cout.setf( ios::showpoint | ios::showpos );
```

带两个 `long` 型参数的 `setf` 成员函数, 例如:

```
cout.setf (ios:: left, ios:: adjustfield);
```

首先清除 `ios::adjustfield` 位, 然后设置 `ios:: left` 标志。该版本的 `setf` 用于与 `ios:: basefield` (用 `ios::dec`、`ios::oct` 和 `ios::hex` 表示)、`ios::floatfield` (用 `ios::scientific` 和 `ios::fixed` 表示) 和 `ios:: adjustfield` (用 `ios:: left`、`ios::right` 和 `ios::internal` 表示) 相关的位段。

成员函数 `unsetf` 清除指定的标志并返回清除前的标志值。

## 11.8 流错误状态

可以用类 `ios` 中的位测试流的状态。类 `ios` 是输入/输出类 `istream`、`ostream` 和 `iostream` 的基类。

当遇到文件结束符时, 输入流中自动设置 `eofbit`。可以在程序中使用成员函数 `eof` 确定是否已经到达文件尾。如果 `cin` 遇到了文件结束符, 那么函数调用:

```
cin.eof()
```

返回 `true`, 否则返回 `false`。

当流中发生格式错误时, 虽然会设置 `failbit`, 但是字符并未丢失。成员函数 `fail` 判断流操作是否失败, 这种错误通常可修复。

当发生导致数据丢失的错误时, 设置 `badbit`。成员函数 `bad` 判断流操作是否失败, 这种严重错误通常不可修复。

如果 `eofbit`、`failbit` 或 `badbit` 都没有设置, 则设置 `goodbit`。

如果函数 `bad`、`fail` 和 `eof` 全都返回 `false`, 则成员函数 `good` 返回 `true`。程序中应该只对“好”的流进行 I/O 操作。

成员函数 `rdstate` 返回流的错误状态。例如, 函数调用 `cout.rdstate` 将返回流的状态, 随后可以用一条 `switch` 语句测试该状态, 测试工作包括检查 `ios:: eofbit`、`ios:: badbit`、`ios:: failbit` 和 `ios:: goodbit`。测试流状态的较好方法是使用成员函数 `eof`、`bad`、`fail` 和 `good`, 使用这些函数不需要程序员熟知特定的状态位。

成员函数 `clear` 通常用于把一个流的状态恢复为“好”, 从而可以对该流继续执行 I/O 操作。由于 `clear` 的默认参数为 `ios:: goodbit`, 所以下列语句:



```
cin.clear();
```

清除 cin，并为流设置 goodbit。下列语句：

```
cin.clear(ios::failbit)
```

实际上给流设置了 failbit。在用自定义类型对 cin 执行输入操作或遇到问题时，用户可能需要这么做。clear 这个名字用在这里似乎并不合适，但规定就是如此。

图 11.29 中的程序演示了成员函数 rdstate、eof、fail、bad、good 和 clear 的使用。

只要 badbit 和 failbit 中有一个被置位，成员函数 operator! 就返回 true。只要 badbit 和 failbit 中有一个被置位，成员函数 operator void\* 就返回 false。这些函数可用于文件处理过程中测试选择结构或循环结构条件的 true/false 情况。

```
1 // Fig. 11.29: fig11_29.cpp
2 // Testing error states.
3 #include <iostream.h>
4
5 int main()
6 {
7     int x;
8     cout << "Before a bad input operation:"
9         << "\ncin.rdstate(): " << cin.rdstate()
10        << "\n    cin.eof(): " << cin.eof()
11        << "\n    cin.fail(): " << cin.fail()
12        << "\n    cin.bad(): " << cin.bad()
13        << "\n    cin.good(): " << cin.good()
14        << "\n\nExpects an integer, but enter a character: ";
15     cin >> x;
16
17     cout << "\nEnter a bad input operation:"
18         << "\ncin.rdstate(): " << cin.rdstate()
19         << "\n    cin.eof(): " << cin.eof()
20         << "\n    cin.fail(): " << cin.fail()
21         << "\n    cin.bad(): " << cin.bad()
22         << "\n    cin.good(): " << cin.good() << "\n\n";
23
24     cin.clear();
25
26     cout << "After cin.clear()"
27         << "\ncin.fail(): " << cin.fail()
28         << "\ncin.good(): " << cin.good() << endl;
29     return 0;
30 }
```

#### 输入结果：

Before a bad input operation:

```
cin.rdstate(): 0
    cin.eof(): 0
    cin.fail(): 0
    cin.bad(): 0
    cin.good(): 0
```

Expects an integer, but enter a character: A

After a bad input operation:

```
cin.rdstate(): 2
cin.eof(): 0
cin.fail(): 2
cin.bad(): 0
cin.good(): 0
```

```
After cin.clear()
cin.fail(): 0
cin.good(): 1
```

图 11.29 测试错误状态

## 11.9 把输出流连到输入流上

交互式应用程序通常要分别用类 `istream` 和 `ostream` 输入和输出数据。当提示信息出现在屏幕上时, 用户输入一个数据来响应。显然, 提示信息必须在执行输入操作前出现。在有输出缓冲区的情况下, 只有在缓冲区已满时、在程序中明确地刷新输出缓冲区时或因程序结束而自动刷新输出缓冲区时, 输出信息才会显示到屏幕上。为保证输出要在下一个输入前显示, C++ 提供了成员函数 `tie`, 该函数可以实现输入/输出操作的同步, 也就是把这两个操作“系在一起”。例如, 函数调用:

```
cin.tie(&cout);
```

把 `cout` (`ostream` 的对象) 连到 `cin` (`istream` 的对象) 上。实际上, C++ 为了建立用户的标准输入/输出环境而自动执行了这个操作, 所以这个专门的调用其实是多余的。但是, 用户可以显式地把其他配对的输入/输出流“系”在一起。使用函数调用:

```
inputStream.tie( 0 );
```

可以从一个输出流中解开输入流 `inputStream`。

## 小结

- I/O 操作是以对数据类型敏感的方式执行的。
- C++ 的 I/O 操作是以字节流的形式实现的。流实际上就是字节序列。
- 输入/输出系统的任务实际上就是以一种稳定、可靠的方式在设备与内存之间传输数据。
- C++ 提供了低级和高级 I/O 功能。低级 I/O 功能通常只在设备和内存之间传输一些字节。高级 I/O 功能把若干个字节组合成有意义的单位, 如整数、浮点数、字符、字符串以及用户自定义类型的数据。
- C++ 提供了无格式 I/O 和格式化 I/O 两种操作。无格式 I/O 传输速度快, 但使用起来较为麻烦。格式化 I/O 按不同的类型对数据进行处理, 但需要增加额外的处理时间, 不适于处理大容量的数据传输。
- 头文件 `iostream.h` 包括了操作所有输入/输出流所需的基本信息, 因此在大多数 C++ 程序中都应该包含这个头文件。
- 在执行格式化 I/O 时, 如果流中带有含参数的流操纵算子, 头文件 `iomanip.h` 所包含的信息是有用的。
- 头文件 `fstream.h` 包含了文件处理操作所需的信息。

- 类 `istream` 支持流的输入操作。
- 类 `ostream` 支持流的输出操作。
- 类 `iostream` 同时支持流的输入、输出操作。
- 类 `istream` 和 `ostream` 是通过单一继承从基类 `ios` 派生出来的。
- 类 `iostream` 是通过多重继承从类 `istream` 和 `ostream` 派生出来的。
- 重载的左移位运算符 (`<<`) 表示流的输出, 称为流插入运算符。
- 重载的右移位运算符 (`>>`) 表示流的输入, 称为流读取运算符。
- `cin` 是类 `istream` 的对象, 它与标准输入设备 (通常指键盘) “连” 在一起。
- `cout` 是类 `ostream` 的对象, 它与标准输出设备 (通常指显示器) “连” 在一起。
- `cerr` 是类 `ostream` 的对象, 它与标准错误输出设备 “连” 在一起。对象 `cerr` 的输出是非缓冲输出, 每一条出错信息都会被立即显示出来。
- 流操纵算子 `endl` 用来插入一个换行符, 并同时刷新输出缓冲区。
- C++ 编译器能够自动识别输入、输出数据的类型。
- 默认情况下, 地址是以十六进制格式显示的。
- 可以将指针强制转换为 `void *` 类型从而输出指针变量中的地址。
- 成员函数 `put` 用于输出单个字符, 可以连续调用 `put` 函数。
- 流的输入通过流读取运算符 `>>` 实现, 该运算符自动跳过输入流中的空白字符。
- 当遇到输入流中的文件结束符时, 运算符 `>>` 返回 `false`。
- 当输入出现格式错误时, 流读取运算符就会设置输入流的 `failbit` 状态位; 如果操作失败则设置 `badbit` 状态位。
- 我们通常在 `while` 循环结构的首部用流读取运算符输入一系列值。当遇到文件结束符时, 流读取运算符返回 0。
- 不带参数的 `get` 函数读取 (输入) 一个字符, 并返回该字符; 当遇到输入流中的文件结束符时, 返回 `EOF`。
- 带一个字符型参数的 `get` 成员函数输入一个字符。当遇到文件结束符时, 函数返回 `EOF`, 否则返回对 `istream` 对象的引用, 并用该引用再次调用 `get` 函数。
- 带有三个参数的 `get` 成员函数, 其三个参数分别是接收字符的字符数组、字符数组的大小和分隔符 (默认值为 `'\n'`)。函数或者在读取比指定的最大字符数少一个字符后结束, 或者在遇到分隔符时结束。空字符终止输入的字符串。函数不把分隔符放到字符数组中, 但是分隔符仍然会保留在输入流中。
- 成员函数 `getline` 与带三个参数的 `get` 函数类似。`getline` 要去除输入流中的分隔符, 但是不把它存放在字符串中。
- 成员函数 `ignore` 用于在需要时跳过流中指定数量的字符 (默认个数是 1), 或在遇到指定的分隔符 (默认分隔符是 `EOF`) 时结束。
- 成员函数 `putback` 将最后一次用 `get` 从输入流中读取的字符放回到输入流中。
- `peek` 成员函数从输入流返回下一个字符, 但不从流中删除这个字符。
- C++ 使用的是类型安全的 I/O 操作。如果 `>>` 或 `<<` 运算符接收到了非法的数据类型, 那么系统将设置各种错误标志位, 用户可根据这些标志位来判断操作 I/O 的成功与失败。
- 调用成员函数 `read`、`write` 可实现无格式输入/输出。这两个函数分别把一定量的字节输入内存和从内存中输出。这些字节都是未经任何格式化的, 仅仅是以原始数据形式输入或输出。
- 成员函数 `gcount` 返回函数 `read` 从输入流中读取的字符个数。

- 成员函数 `read` 把指定个数的字符输入到字符数组中。如果读取的字符个数少于指定的数目, 就设置标志位 `failbit`。
- 如下方法可改变流中整数的基数: 使用流操纵算子 `hex` 可设置十六进制基数 (基数为 16)、使用流操纵算子 `oct` 可设置八进制基数 (基数为 8)、使用流操纵算子 `dec` 可恢复十进制基数。只有显式地改变流的基数, 否则流的基数是不变的。
- 流操纵算子 `setbase` 也可以为整数输出设置基数。流操纵算子 `setbase` 带有一个整数参数 10、8 或 16。
- 流操纵算子 `setprecision` 或成员函数 `precision` 控制浮点数小数点后面的位数。设置了精度以后, 该精度对之后所有的输出操作都有效, 直到再一次设置精度为止。无参数的成员函数 `precision` 返回当前的精度。
- 使用任何参数化的流操纵算子都必须在程序中包含头文件 `iomanip.h`。
- 成员函数 `width` 设置当前域宽并返回以前设置的域宽。如果显示数据所需的宽度比设置的域宽小, 空位用填充字符填充。域宽设置仅对下一次流读取或流插入操作有效; 然后域宽隐式设为 0 (可以输出任意长度的序列值)。长度大于域宽的值也可以完整输出。不带参数的函数 `width` 返回当前域宽。流操纵算子 `setw` 也可以用来设置域宽。
- 在输入操作中, 流操纵算子 `setw` 用于设置字符串长度的最大值。如果输入了一个长度大于最大值的字符串, 那么该字符串将被分为若干个长度不大于限定值的字符串。
- 用户可以建立自己的流操纵算子。
- 成员函数 `setf`、`unsetf` 和 `flags` 控制标志设置。
- `skipws` 标志指示 >> 跳过输入流中的空白字符。流操纵算子 `ws` 还可用于跳过输入流中的前导空白字符。
- 格式标志在类 `ios` 中被定义为枚举值。
- 虽然成员函数 `flags`、`setf` 和 `unsetf` 可以控制格式标志, 但是许多程序员更喜欢使用流操纵算子。程序员可以用按位或操作 (`|`) 把各种标志选项结合在一个 `long` 类型的值中。调用成员函数 `flags` 并指定要进行或操作的标志选项就可以在流中设置这些标志, 并返回一个包括以前标志选项的 `long` 类型的值。返回值通常要保存起来, 以便使用保存值调用 `flags` 来恢复以前的选项。
- `flags` 函数必须指定一个代表所有标志设置的值。而带有一个参数的函数 `setf` 将指定的标志与现存的标志设置进行或操作, 形成新的格式状态。
- 设置 `showpoint` 标志是为了强制输出浮点数的小数点和由精度指定的有效数字的个数。
- `left` 标志可以使输出域左对齐并把填充字符放在输出数据的右边, `right` 标志可以使输出域右对齐并把填充字符放在输出数据的左边。
- `internal` 标志指示将一个数的符号位 (设置了 `iso::showbase` 标志时为基数) 在域宽内左对齐, 数值右对齐, 中间的空白由填充字符填充。
- `left`、`right` 和 `internal` 标志包含在 `iso::adjustfield` 中。
- 成员函数 `fill` 用于设置与 `left`、`right` 和 `internal` 一起使用的使输出域对齐的填充字符; 如果没有特别指定, 空格即为填充字符。 `fill` 函数返回以前设置的填充字符。流操纵算子 `setfill` 同样也可设置填充字符。
- 静态成员 `ios::basefield` 包括 `oct`、`hex` 和 `dec` 位, 这些位分别指定把整数作为八进制、十六进制和十进制值处理。如果没有设置这些位, 则流插入运算默认整数为十进制数, 流读取运算按提供的方式处理数据。

- 设置 `showbase` 标志可强制输出整数值的基数。
- `ios::scientific` 和 `ios::fixed` 标志包含在静态数据成员 `ios::floatfield` 中。这些标志用于控制浮点数的输出格式。设置 `scientific` 标志使浮点数按照科学记数法格式输出, 设置 `fixed` 标志使浮点数按照 `precision` 成员函数指定的精度输出。
- 函数调用 `cout.setf(0, ios::floatfield)` 恢复浮点数的默认显示格式。
- 设置 `ios::uppercase` 标志用来使大写的 X 和 E 分别同十六进制整数和以科学记数法表示的浮点数一起输出。一旦设置 `ios::uppercase` 标志, 十六进制数中的所有字母都转换成大写。
- 无参数的成员函数 `flags` 只返回格式标志的当前设置 (long 类型的值)。带一个 long 类型参数的成员函数 `flags` 按参数指定的格式设置标志并返回以前的标志设置。
- 成员函数 `setf` 设置参数所指定的格式标志, 并返回前一次的标志设置值 (long 类型的值)。
- 成员函数 `setf(long setBits, long resetBits)` 清除 `resetBits` 中的位, 然后设置 `setBits` 位。
- 成员函数 `unsetf` 清除已设置的标志并返回清除前的标志值。
- 参数化的流操纵算子 `setiosflags` 与成员函数 `flags` 执行同样的功能。
- 参数化的流操纵算子 `resetiosflags` 与成员函数 `unsetf` 执行同样的功能。
- 流的状态可以用类 `ios` 中的位来检测。
- 当遇到文件结束符时, 输入流中会被自动设置 `eofbit`。可以在程序中用成员函数 `eof` 确定是否设置了 `eofbit`。
- 当流中发生格式错误时虽然会设置 `failbit`, 但是字符并未丢失。成员函数 `fail` 判断流操作是否失败, 这种错误通常可恢复。
- 当发生导致数据丢失的流格式错误时, 设置 `badbit`。成员函数 `bad` 判断流操作是否失败, 这种严重错误通常不可修复。
- 如果函数 `bad`、`fail` 和 `eof` 全都返回 `false`, 则成员函数 `good` 返回 `true`。应该只对“好”的流进行 I/O 操作。
- 成员函数 `rdstate` 返回流的错误状态。
- 成员函数 `clear` 通常用于把一个流的状态恢复为“好”, 从而可以对该流继续执行 I/O 操作。
- 为保证输出在下一个输入前显示, C++ 提供了成员函数 `tie`, 该函数可以实现输入/输出操作的同步。

## 术语

`bad member function` `bad` 成员函数

`badbit`

`cerr`

`cin`

`clear member function` `clear` 成员函数

`clog`

`cout`

`dec stream manipulator` `dec` 流操纵算子

`default fill character (space)` 默认填充字符 (空格)

`default precision` 默认精度

`end-of-file` 文件尾

`endl`

`eof member function` `eof` 成员函数

`eofbit`

`extensibility` 可扩展性

`fail member function` `fail` 成员函数

`failbit`

`field width` 域宽

`fill character` 填充字符

`fill member function` `fill` 成员函数

- flags member function flags 成员函数
- flush member function flush 成员函数
- flush stream manipulator flush 流操纵算子
- format flags 格式标志
- format states 格式状态
- formatted I/O 格式化 I/O
- fstream classfstream 类
- gcount member function gcount 成员函数
- get member function get 成员函数
- getline member function getline 成员函数
- good member function good 成员函数
- hex stream manipulator hex 流操纵算子
- ifstream class ifstream 类
- ignore member function ignore 成员函数
- in-core formatting 内核格式化
- in-memory formatting 内存格式化
- <iomanip.h> standard header file <iomanip.h>标准头文件
- ios class ios 类
- ios::adjustfield
- ios::basefield
- ios::fixed
- ios::floatfield
- ios::internal
- ios::scientific
- ios::showbase
- ios::showpoint
- ios::showpos
- iostream class iostream 类
- istream class istream 类
- leading 0 (octal) 八进制值的开头
- leading 0x or 0X (hexadecimal) 十六进制值的开头
- left 左边
- left-justified 左对齐
- oct stream manipulator oct 流操纵算子
- ofstream class ofstream 类
- operator! member function operator! 成员函数
- operator void \* member function operator void \* 成员函数
- ostream class ostream 类
- padding 填充
- parameterized stream manipulator 参数化流操纵算子
- peek member function peek 成员函数
- precision member function precision 成员函数
- predefined streams 预定义的流
- put member function put 成员函数
- putback member function putback 成员函数
- rdstate member function rdstate 成员函数
- read member function read 成员函数
- resetiosflags stream manipulator resetiosflags 流操纵算子
- right-justified 右对齐
- setbase stream manipulator setbase 流操纵算子
- setf member function setf 成员函数
- setfill stream manipulator setfill 流操纵算子
- setiosflags stream manipulator setiosflags 流操纵算子
- setprecision stream manipulator setprecision 流操纵算子
- setw stream manipulator setw 流操纵算子
- skipws
- stream class libraries 流类库
- stream-extraction operator (>>) 流读取运算符
- stream input 流输入
- stream-insertion operator (<<) 流插入运算符
- stream manipulator stream 流操纵算子
- stream output 流输出
- tie member function tie 成员函数
- type-safe I/O 类型安全 I/O
- unformatted I/O 无格式 I/O
- unsetf member function unsetf 成员函数
- uppercase 大写
- user-defined streams 用户自定义流
- whitespace characters 空白字符
- width 宽度
- write member function write 成员函数
- ws member function ws 成员函数

## 自测练习

## 11.1 填空

- a) 重载的流运算符函数经常定义为类的 \_\_\_\_\_ 函数。
- b) 能够设置的格式对齐位包括 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- c) C++ 中的输入 / 输出是以字节 \_\_\_\_\_ 的形式实现的。
- d) 参数化的流操纵算子 \_\_\_\_\_ 和 \_\_\_\_\_ 用于设置和清除格式化状态标志。
- e) 大多数 C++ 程序都要包含 \_\_\_\_\_ 头文件。该文件中包含了所有输入 / 输出流操作所需的基本信息。
- f) 成员函数 \_\_\_\_\_ 和 \_\_\_\_\_ 用于设置和清除格式化状态标志。
- g) 头文件 \_\_\_\_\_ 中包含了执行内存格式化所需的信息。
- h) 当使用带参数的操纵算子时, 程序中必须包含头文件 \_\_\_\_\_。
- i) 头文件 \_\_\_\_\_ 中包含了处理用户控制的文件操作所需的信息。
- j) 流操纵算子 \_\_\_\_\_ 向输出流中插入一个换行符并刷新输出缓冲区。
- k) 头文件 \_\_\_\_\_ 中包含了用 C、C++ 语言式 I/O 混合编程所需的信息。
- l) 类 `ostream` 的成员函数 \_\_\_\_\_ 用于执行无格式输出。
- m) 类 \_\_\_\_\_ 支持输入操作。
- n) 标准错误流的输出发送给流对象 \_\_\_\_\_ 或 \_\_\_\_\_。
- o) 类 \_\_\_\_\_ 支持输出操作。
- p) 流插入运算符是 \_\_\_\_\_。
- q) 与系统中标准设备对应的 4 个对象是 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- r) 流读取运算符是 \_\_\_\_\_。
- s) 流操纵算子 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_ 分别指定整数按八进制、十六进制、十进制格式显示。
- t) 浮点数的默认精度值是 \_\_\_\_\_。
- u) 设置 \_\_\_\_\_ 标志位可使显示的正数前面带有一个加号。

## 11.2 判断下列说法是否正确。如果不正确, 请说明原因。

- a) 带一个 `long` 类型参数的流成员函数 `flags()` 按参数值设置 `flags` 状态变量, 并返回以前设置的标志值。
- b) 重载流插入运算符 `<<` 和流读取运算符 `>>` 是为了处理所有的标准数据类型, 包括字符串、内存地址 (只能用流插入运算符) 和所有用户自定义的数据类型。
- c) 无参数的成员函数 `flags()` 可以设置状态变量 `flags` 中的所有状态位。
- d) 流读取运算符的参数是对 `istream` 对象的引用和对自定义类型对象的引用, 返回对 `istream` 对象的引用。
- e) 流操纵算子 `ws` 可跳过输入流中的前导空白字符。
- f) 流插入运算符 `<<` 的两个参数是对 `istream` 对象的引用和对自定义类型对象的引用, 返回对 `istream` 对象的引用。
- g) 用流读取运算符 `>>` 进行输入操作总会跳过输入流中的前导空白字符。
- h) 输入、输出的特性是由 C++ 本身所提供的。
- i) 流成员函数 `rdstate()` 返回当前流的状态。
- j) `cout` 流通常是与显示器相连的。

- k) 如果成员函数 `bad()`、`fail()` 和 `eof()` 都返回 `false`，则流成员函数 `good()` 返回 `true`。
- l) `cin` 流通常是与显示器相连的。
- m) 如果在流操作期间发生了不可恢复的致命错误，成员函数 `bad()` 返回 `true`。
- n) 到 `cerr` 的输出是非缓冲输出，到 `clog` 的输出是缓冲输出。
- o) 当设置 `ios::showpoint` 标志位时，浮点数被强制以默认精度格式（6位小数位）输出（假定未改变精度值使浮点数按指定精度输出）。
- p) 类 `ostream` 的成员函数 `put` 用于输出指定数目的字符。
- q) 流操纵算子 `dec`、`oct`、`hex` 只对下一个输出整数有效。
- r) 在输出操作中，内存地址是以 `long` 类型表示的。

### 11.3 用一条 C++ 语句实现下述要求。

- a) 输出字符串 "Enter your name:"。
- b) 设置一个标志，使科学记数法中的指数以及十六进制数中的字母按大写的格式输出。
- c) 输出 `char *` 类型变量 `string` 的地址。
- d) 设置一个标志，以科学记数法显示浮点数。
- e) 输出 `int *` 类型变量 `integerPtr` 的地址。
- f) 设置一个标志，使得在输出整数时，八进制数和十六进制数显示出其基数。
- g) 输出 `float *` 类型变量 `floatPtr` 所指向的值。
- h) 当所设置的域宽长度大于输出数据所需宽度时，用成员函数设置填充字符 '\*'。再写一条用流操纵算子实现该功能的语句。
- i) 用类 `ostream` 的函数 `put` 在一条语句中输出字符 'O'、'K'。
- j) 从输入流中获取下一个字符，但并不提取它。
- k) 采用两种不同的方法，用类 `istream` 的成员函数 `get` 向 `char` 类型变量 `c` 输入一个字符。
- l) 输出并删除输入流中的下 6 个字符。
- m) 用类 `istream` 的成员函数 `read` 给 `char` 类型数组 `line` 输入 50 个字符。
- n) 读入 10 个字符到字符数组 `name` 中，当遇到分隔符 '.' 时结束读操作，但并不删除输入流中的分隔符。另外再写一条语句完成上述功能，但需要删除输入流中的分隔符。
- o) 用类 `istream` 的成员函数 `gcount` 统计字符型数组 `line` 中的字符个数（`line` 中的字符是通过调用类 `istream` 的成员函数 `read` 来输入的），然后根据统计的字符个数，用类 `ostream` 的成员函数 `write` 输出 `line` 中的字符。
- p) 分别编写两条语句，用成员函数和流操纵算子刷新输出缓冲区。
- q) 输出下列值：124、18.376、'Z'、1000000 和 "String"。
- r) 用成员函数输出当前浮点数的精度。
- s) 给 `int` 类型变量 `months` 输入一个整数值，给 `float` 类型变量 `percentageRate` 输入一个浮点数。
- t) 用流操纵算子输出 1.92、1.925 和 1.9258，精度是三位小数位。
- u) 用流操纵算子分别按八进制、十六进制、十进制格式输出整数 100。
- v) 按八进制、十六进制、十进制格式输出整数 100，要求用同一个流操纵算子来改变整数的基数。
- w) 按右对齐方式、以 10 位域宽输出 1234。
- x) 把字符读入字符数组 `line` 中，当遇到指定的分隔符 'z' 时或读取的字符个数达到限定值 20（包括空字符）时，停止读取操作，该语句不从输入流中读取分隔符。



y) 按域宽 x、精度 y (x、y 为整型变量) 输出 double 类型值 87.4573。

11.4 指出并纠正下列语句中的错误。

a) `cout << "Value of x <= y is: "<< x <= y;`

b) 下面的语句要输出字符 'c' 的整数值:

```
cout<< 'c';
```

c) `cout << ""A string in quotes"";`

11.5 写出下面语句的输出结果。

a) `cout << "12345" << endl;`

```
cout.width( 5 );
```

```
cout.fill('*');
```

```
cout << 123 << endl<< 123;
```

b) `cout << setw( 10 ) << setfill( '$' ) << 10000;`

c) `cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;`

d) `cout << setiosflags( ios::showbase ) << oct << 99`  
`<< endl << hex << 99;`

e) `cout << 100000 << endl`

```
<< setiosflags( ios::showpos ) << 100000;
```

f) `cout << setw( 10 ) << setprecision( 2 ) <<`

```
<< setiosflags( ios::scientific ) << 444.93738;
```

## 自测练习答案

11.1 a) 友元。b) `ios::left`、`ios::right`和`ios::internal`。c) 流。d) `setiosflags`、`resetiosflags`。e) `iostream.h`。  
f) `setw`、`unsetf`。g) `strstream.h`。h) `iomanip.h`。i) `fstream.h`。j) `endl`。k) `stdiostream.h`。l) `write`。  
m) `istream`。n) `cerr`、`clog`。o) `ostream`。p) `<<`。q) `cin`、`cout`、`cerr`和`clog`。r) `>>`。s) `oct`、`hex`、`dec`。t) 6位精度。u) `ios::showpos`。

11.2 a) 正确。

b) 不正确。重载的流插入运算符和流读取运算符不能用于所有的用户自定义类型。程序员必须为每一个用户自定义类型提供重载该运算符的运算符函数。

c) 不正确。无参数的成员函数 `flags()` 只返回状态变量 `flags` 的当前值。

d) 正确。

e) 正确。

f) 不正确。如果要重载流插入运算符,流插入运算符的两个参数必须是对 `ostream` 对象的引用和对自定义类型的对象的引用,返回对 `ostream` 对象引用。

g) 正确。除非 `ios::skipws` 未置位。

h) 不正确。C++ 的输入、输出特性是 C++ 标准库的一部分, C++ 语言本身并不包括输入、输出和文件处理能力。

i) 正确。

j) 正确。

k) 正确。

l) 不正确。`cin` 流通常是与计算机的键盘相连。

m) 正确。

n) 正确。

o) 正确。

- p) 不正确。类 `ostream` 的成员函数 `put` 用于输出单个的字符。
- q) 不正确。流操纵算子 `dec`、`oct`、`hex` 用于设置输出整数的基数，直到下一次设置时或者在程序终止时才会改变。
- r) 不正确。在输出操作中，内存地址的默认显示方式是十六进制格式。如果要按 `long` 类型显示，则必须先转换成对应于 `long` 类型的值。
- 11.3 a) `cout << "Enter your name: ";`  
 b) `cout.setf(ios::uppercase);`  
 c) `cout << (void *)string`  
 d) `cout.setf(ios::scientific,ios::floatfield);`  
 e) `cout << integerPtr;`  
 f) `cout << setiosflags(ios::showbase);`  
 g) `cout << * floatPtr;`  
 h) `cout.fill( '*' );`  
    `cout << setfill( '*' );`  
 i) `cout.put( 'O' ) .put( 'K' );`  
 j) `cin.peek();`  
 k) `c = cin.get();`  
    `cin.get ( c );`  
 l) `cin.ignore( 6 );`  
 m) `cin.read( line,50 );`  
 n) `cin.get( name, 10, '.' );`  
    `cin.getline( name, 10, '.' );`  
 o) `cout.write( line, cin.gcount() );`  
 p) `cout.flush();`  
    `cout << flush;`  
 q) `cout << 124 << 18.376 << 'z' << 1000000 << "String";`  
 r) `cout << cout.precision();`  
 s) `cin >> months >> percentageRate;`  
 t) `cout << setprecision( 3 ) << 1.92 << '\t'`  
    `<< 1.925 << '\t'<< 1.9258;`  
 u) `cout << oct << 100 << hex << 100 << dec << 100;`  
 v) `cout << 100 << setbase( 8 ) << 100 << setbase( 16 ) << 100;`  
 w) `cout << setw( 10 ) << 1234;`  
 x) `cin.get( line, 20, 'z' );`  
 y) `cout <<setw( x ) << setprecision( y ) << 87.4573;`

- 11.4 a) 不正确：运算符 `<<` 的优先级比 `<=` 高，语句求值不正确，因此编译器会报告错误。  
 纠正：在表达式 `x<=y` 两边加上括号。
- b) 不正确：不能像在 C 语言中那样在 C++ 中直接输出字符的 ASCII 码值。  
 纠正：如果需要输出某一字符的 ASCII 值，必须先计算出它的 ASCII 值再输出，例如：
- ```
cout << int('C');
```

- c) 不正确：除非使用转义序列来加以区别，否则无法在字符串中输出双引号 ""。’  
纠正：可用下面两种方法之一输出双引号 ""：

```
cout << '"' << "A string in quotes" << "'";  
cout << "\"A string in quotes\"";
```

- 11.5 a) 12345  
      \*\*123  
      123  
      b) \$\$\$\$\$10000  
      c) 1024.988  
      d) 0143  
          0x63  
      e) 100000  
          +100000  
      f) 4.45e+02

## 练习

- 11.6 用一条 C++ 语句实现下述要求：
- a) 以左对齐方式输出整数 40000，域宽为 15。
  - b) 把一个字符串读到字符型数组变量 state 中。
  - c) 打印有符号数 200 和无符号数 200。
  - d) 将十进制整数 100 以 0x 开头的十六进制格式输出。
  - e) 把字符读到数组 s 中，直到遇到字符 'p' 或读取的字符个数达到限定值 10 时终止读取操作。同时从输入流中读取分隔符并删除。
  - f) 用前导 0 格式打印 1.234，域宽为 9。
  - g) 从标准输入流中读取字符串 "characters"，将其存储在字符数组 s 中。读取过程中去掉双引号，读取字符个数的最大限定值为 50（包括空字符）。
- 11.7 编写一个程序，测试十进制、八进制、十六进制格式整数值的输入，分别按三种不同的基数输出。测试数据为 10、010、0x10。
- 11.8 编写一个程序，用强制类型转换运算符把指针值转换成各种整数数据类型后，打印出指针值。哪种数据类型打印出奇怪的值？哪种数据类型产生了错误？
- 11.9 编写一个程序，分别用不同的域宽打印出整数 12345 和浮点数 1.2345。当域宽小于数值的实际需要的域宽时会发生什么情况？
- 11.10 编写一个程序，将 100.453627 取整到最近似的个位、十分位、百分位、千分位和万分位，打印出结果。
- 11.11 编写一个程序，从键盘输入一个字符串，判断字符串的长度，然后以字符串长度的两倍作为域宽打印出该字符串。
- 11.12 编写一个程序，将华氏温度 0 度 ~212 度转换为浮点型摄氏温度，浮点数精度为 3。转换公式如下：

```
celsius = 5.0/9.0 * (fahrenheit - 32);
```

输出用两个右对齐列，摄氏温度前面加上正负号。

- 11.13 在不同的编程语言中,字符串有的是用单引号''括起,有的是用双引号"括起。编写一个程序,读取三个不同的字符串 suzy、"suzy"和'suzy'。看看单引号和双引号是被忽略了,还是被当作字符串的一部分一起读取了。
- 11.14 图 8.3 的程序为类 PhoneNumber 的输入和输出对象而重载了流读取运算符和流插入运算符。重新编写流读取运算符,使它能够检测下面的输入错误。注意,函数 operator>> 的代码全部需要重写。
- a) 输入一个完整的电话号码到一个字符数组中,检测输入的字符总数,例如电话号码 (800)555-1212 的字符总数为 14 个。如果输入错误,用流成员函数 clear 设置 ios::failbit 位。
  - b) 电话号码中的区号和局号不能以 0 或 1 打头。检测区号和局号的第一位,判断是否为 0 或 1,若是,则用流成员函数 clear 设置 ios::failbit 位。
  - c) 区号的中间一位是 0 或 1,检测中间位,判断是否为 0 或 1,如果输入错误,则用流成员函数 clear 设置 ios::failbit 位。如果输入正确,即 ios::failbit 位为 0,则把电话号码的三个部分分别复制到对象 PhoneNumber 的成员 areacode、exchange 和 line 中。在主程序里,如果置位 ios::failbit,则程序打印出一条出错消息,并且不打印电话号码而结束运行。
- 11.15 编写完成下列要求的程序:
- a) 建立用户自定义类 Point,该类包含 private 整数数据成员 xCoordinate 和 yCoordinate,并在类中声明了重载的流插入和流读取运算符函数为其友元。
  - b) 定义流插入和流读取运算符函数。流读取运算符函数判断输入的数据是否合法,如果是非法数据,则置位 ios::failbit 以指示输入不正确。发生错误后,流插入运算符函数不显示 Point 的对象(点)的坐标信息。
  - c) 编写函数 main,用重载的流插入运算符和流读取运算符测试自定义类 Point 的输入和输出。
- 11.16 编写完成下列要求的程序:
- a) 建立用户自定义复数类 Complex,类中包括 private 整数数据成员 real 和 imaginary,并声明了重载的流插入和流读取运算符函数为其友元。
  - b) 定义流插入和流读取运算符函数。流读取运算符函数判断输入的数据是否合法,如果是非法数据,则置位 ios::failbit 以指示输入不正确。必须按以下格式输入:  
$$3 + 8i$$
  - c) 数据可为整数或负数,还可以只给其中一个数赋值。未给出的数值,由相应的数据成员设置为 0。当发生输入错误时,流插入运算符不显示该复数值。输出格式与上述的输入格式相同,对于为负的虚部要打印出负号。
  - d) 编写函数 main,用重载的流插入运算符和流读取运算符测试自定义类 Complex 的输入和输出。
- 11.17 用 for 结构为 ASCII 字符集中 ASCII 码值从 33~126 的字符打印出一张 ASCII 码表。要求输出十进制值、八进制值、十六进制值和 ASCII 码值,并在程序中使用流操纵算子 dec、oct 和 hex。
- 11.18 编写一个程序,用成员函数 getline 和带三个参数的成员函数 get 输入带有空字符的字符串。get 函数不读取分隔符,分隔符仍保留在输入流中,并让 getline 从输入流中读取并删除分隔符。把未读取的字符留在输入流中会发生什么情况?

- 11.19 编写一个程序，建立用户自定义流操纵算子 `skipwhite` 以跳过输入流中的前导空白字符。该流操纵算子使用 `ctype.h` 函数库中的 `isspace` 函数测试输入的字符是否是空白字符。要求用 `istream` 的成员函数 `get` 来读取每一个字符，当遇到一个非空白字符时，流操纵算子 `skipwhite` 把该字符放回到输入流中，返回对 `istream` 对象的引用。

建立一个 `main` 函数，测试自定义流操纵算子（在 `main` 函数中清除 `ios::skipws` 标志位，以保证流读取运算符无法自动跳过空白字符）。然后输入一个以空白字符开头的字符，对流操纵算子进行测试，读取完毕之后，输出所读取的字符以证实空白字符确实未被输入。

# 第12章 模 板

## 教学目标

- 用函数模板生成相关（重载）函数组
- 区分函数模板与模板函数
- 用类模板生成相关类型组
- 区分类模板与模板类
- 了解如何重载模板函数
- 了解模板、友元、继承与静态成员之间的关系

## 12.1 简介

本章介绍 C++ 最强大的特性之一——模板。模板使我们可以用一个代码段指定一组相关（重载）函数（称为模板函数）或一组相关类（称为模板类）。

我们可以对数组排序函数编写一个函数模板，然后让 C++ 自动生成模板函数，可以对 int 数组、float 数组和字符串数组等等进行排序。

第 3 章介绍了函数模板。如果读者没有阅读该章，则这里再提供一些介绍和例子。

我们可以对堆栈类编写一个类模板，然后让 C++ 自动生成如 int、float 和 string 堆栈类的类模板。

注意区分函数模板与模板函数：函数模板和类模板像是具有各种形状的模板，而模板函数和模板类则相当于按照模板描绘，其形状都是相同的，只是画上不同的颜色。

### 软件工程视点 12.1

模板是 C++ 的软件复用的功能之一。

本章介绍一些函数模板和类模板的例子，并介绍模板与其他 C++ 特性（如重载、继承、友元和 static 成员）之间的关系。

这里介绍的模板机制的设计和细节基于 Bjarne Stroustrup 的论文《Parameterized Types for C++》，发表于 1988 年 10 月在科罗拉多州丹佛举办的 USENIX C++ 会议上（Proceedings of the USENIX C++ Conference）。

本章只是关于模板问题的简介，第 20 章“标准模板库（STL）”将深入介绍模板容器类、迭代器和 STL 算法。第 20 章有几十个基于模板的“有生命力的代码”，演示了更复杂的模板编程技术。

## 12.2 函数模板

重载函数通常是基于不同的数据类型完成类似的操作。如果对每种数据类型的操作是相同的，那么用函数模板完成这项工作更为简洁和方便。程序员对函数模板的定义只编写一次。基于调用函

数时提供的参数类型, C++ 自动产生单独的目标代码函数来正确地处理每种类型的调用。在C语言中, 这个任务是用预处理指令 `#define` 建立的宏完成的 (见第17章)。但是, 宏可能会产生副作用, 并且使编译器不能进行类型检查。函数模板和宏一样的简洁, 并且还能让编译器进行全面的类型检查。

#### 测试与调试提示 12.1

函数模板和宏一样允许软件复用。但与宏不同的是, 函数模板还可以消除许多类型错误, 因为C++提供了安全的全面类型检查。

所有的函数模板定义都是用关键字 `template` 开始的, 该关键字之后是用尖括号 `<>` 括起来的形式参数表。每一个形式参数之前都有关键字 `class`, 例如:

```
template< class T >
或
template< class ElementType >
或
template< class BorderType, class FillType >
```

内部类型和自定义类型可用来指定传递给函数的参数类型、函数返回类型和声明函数中变量, 函数模板中的形式参数的用法与之类似。该函数定义的方式与定义其他函数类似。注意关键字 `class` 指定函数模板类型参数, 实际上表示“任何内部类型或用户自定义类型”。

#### 常见编程错误 12.1

函数模板的每个形式类型参数之前不放置关键字 `class` (或新的关键字 `typename`)。

下面看看图 12.1 的 `printArray` 函数模板, 这个函数的用法见图 12.2 的完整程序。

```
1 template< class T >
2 void printArray( const T *array, const int count )
3 {
4     for ( int i = 0; i < count; i++ )
5         cout << array[ i ] << " ";
6
7     cout << endl;
8 }
```

图 12.1 函数模板

该函数模板把惟一的形式参数 `T` (`T` 一般作为类型参数) 声明为函数 `printArray` 打印的数组类型。当编译器检测到程序源代码中调用函数 `printArray` 时, 用 `printArray` 的第一个参数的类型替换掉整个模板定义中的 `T`, 并建立用来打印指定类型数组的一个完整的模板函数, 然后再编译这个新建的函数。图 12.2 的程序演示了三个 `printArray` 函数, 这三个函数分别需要一个 `int` 类型的数组、一个 `double` 类型的数组和一个 `char` 类型的数组。`int` 类型数组的实例函数如下所示:

```
void printArray( const int *array, const int count )
{
    for ( int i = 0; i < count; i++ )
        cout << array[ i ] << " ";
    cout << endl;
}
```

模板函数中的每一个形式参数要在函数参数表中至少出现一次。形式参数的名字可以只在模板函数的形式参数表中出现一次。同一个形式参数名可用于多个模板函数。

图 12.2 的程序反映了模板函数 `printArray` 的用法。程序首先实例化 `int` 数组 `a`、`double` 数组 `b` 和 `char` 数组 `c`，长度分别为 5、7、6。然后调用 `printArray` 打印每个数组，一次用 `a` 的第一个参数，类型为 `int*`；一次用 `b` 的第一个参数，类型为 `double*`；一次用 `c` 的第一个参数，类型为 `char*`。

例如，下列语句：

```
printArray( a, aCount );
```

使编译器实例化 `printArray` 模板函数，类型参数 `T` 为 `int`。下列语句：

```
printArray( b, bCount );
```

使编译器实例化第二个 `printArray` 模板函数，类型参数 `T` 为 `double`。下列语句：

```
printArray( c, cCount );
```

使编译器实例化第三个 `printArray` 模板函数，类型参数 `T` 为 `char`。

本例中，模板机制使程序员不必用下列原型编写三个重载函数：

```
void printArray( const int*, const int );
void printArray( const double*, const int );
void printArray( const char*, const int );

1 // Fig 12.2: fig12_02.cpp
2 // Using template functions
3 #include <iostream.h>
4
5 template< class T >
6 void printArray( const T *array, const int count )
7 {
8     for ( int i = 0; i < count; i++ )
9         cout << array[ i ] << " ";
10
11     cout << endl;
12 }
13
14 int main()
15 {
16     const int aCount = 5, bCount = 7, cCount = 6;
17     int a[ aCount ] = { 1, 2, 3, 4, 5 };
18     double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
19     char c[ cCount ] = "HELLO"; // 6th position for null
20
21     cout << "Array a contains:" << endl;
22     printArray( a, aCount ); // integer template function
23
24     cout << "Array b contains:" << endl;
25     printArray( b, bCount ); // double template function
26
27     cout << "Array c contains:" << endl;
28     printArray( c, cCount ); // character template function
29
30     return 0;
```



```
31 }
```

**输出结果:**

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
HELLO
```

图 12.2 使用模板函数

#### 性能提示 12.1

模板提供了软件复用的好处。请记住, 尽管模板只编写一次, 但程序中仍然实例化多个模板函数和模板类的副本。这些副本会占用大量内存。

## 12.3 重载模板函数

模板函数与重载是密切相关的。从函数模板产生的相关函数都是同名的, 因此编译器用重载的解决方法调用相应函数。

函数模板本身可以用多种方式重载。我们可以提供其他函数模板, 指定不同参数的相同函数名。例如, 图 12.2 的 `printArray` 函数模板可以用另一 `printArray` 函数模板重载, 用参数 `lowSubscript` 和 `highSubscript` 指定要打印的数组部分 (见练习 12.4)。

函数模板也可以用其他非模板函数 (同名而参数不同) 重载。例如, 图 12.2 的 `printArray` 函数模板可以用一个非模板函数重载, 指定以整齐的表格式分栏打印字符串数组 (见练习 12.5)。

#### 常见编程错误 12.2

如果使用用户自定义类的类型调用模板, 而模板对该类型对象使用 `==`、`+`、`<=` 等运算符, 那么这些运算符需要重载。如果不重载这些运算符, 则会发生错误, 因为编译器在这些函数不存在的情况下仍然调用这些重载的运算符函数。

编译器通过匹配过程确定调用哪个函数。首先, 编译器寻找和使用最符合函数名和参数类型的函数调用。如果找不到, 则编译器检查是否可以用函数模板产生符合函数名和参数类型的模板函数。过去, 这种与模板的匹配过程要求所有参数类型都完全匹配, 而不能进行自动转换。现在已经没有这么严格, 可以采用通常的重载规则。

#### 常见编程错误 12.3

编译器通过匹配过程确定调用哪个函数, 如果找不到匹配或产生多个匹配, 就会产生编译错误。

## 12.4 类模板

堆栈独立于栈中数据项的类型, 这一点不难理解。但是, 用程序实现堆栈的时候又必须提供数据类型, 这为实现软件的复用性提供了一次很好的机会。所用的方法是描述一个通常意义上的堆栈, 然后建立这个类的实例类。所建的实例类虽然是通用类的副本, 但是它具有指定的类型。C++ 的模板类提供了这种功能。

## 软件观点 12.2

类模板通过实例化通用类的特定版本提高了软件的复用性。

为了说明如何定制通用类的模板以形成指定的模板类,模板类需要一种或多种类型参数,所以模板类也常常称为参数化类型。

需要生成多种模板类的程序员只需简单地编写一个通用类模板的定义。在需要用模板建立一个新类的时候,程序员只需要用一种简洁的表示方法,编译器就会写出模板类的源代码。例如,堆栈类的模板可以作为编写各种类型堆栈的基础(如 float 类型、int 类型或 char 类型的堆栈等等)。

图 12.3 中的程序定义了 Stack(堆栈)的类模板。模板类与通常的类定义没有什么不同,只是以如下所示的首部开头(第 8 行):

```
template < class T >
```

上述首部指出了这是一个类模板的定义,它有一类型参数 T(表示所要建立的 Stack 类的类型)。程序员不需要专门使用标识符 T,任何标识符都可以使用。Stack 中存储的元素类型在 Stack 类首部和成员函数定义中一般表示为 T。稍后将介绍如何将 T 与特定类型(如 double 或 int)相关联。

```
1 // Fig. 12.3: tstack1.h
2 // Class template Stack
3 #ifndef TSTACK1_H
4 #define TSTACK1_H
5
6 #include <iostream.h>
7
8 template< class T >
9 class Stack {
10 public:
11     Stack( int = 10 );    // default constructor (stack size 10)
12     ~Stack() { delete [] stackPtr; } // destructor
13     bool push( const T& ); // push an element onto the stack
14     bool pop( T& );       // pop an element off the stack
15 private:
16     int size;             // # of elements in the stack
17     int top;              // location of the top element
18     T *stackPtr;          // pointer to the stack
19
20     bool isEmpty() const { return top == -1; } // utility
21     bool isFull() const { return top == size - 1; } // functions
22 };
23
24 // Constructor with default size 10
25 template< class T >
26 Stack< T >::Stack( int s )
27 {
28     size = s > 0 ? s : 10;
29     top = -1;             // Stack is initially empty
30     stackPtr = new T[ size ]; // allocate space for elements
31 }
32
33 // Push an element onto the stack
34 // return true if successful, false otherwise
35 template< class T >
```

```
36 bool Stack< T >::push( const T &pushValue )
37 {
38     if ( !isFull() ) {
39         stackPtr[ ++top ] = pushValue; // place item in Stack
40         return true; // push successful
41     }
42     return false; // push unsuccessful
43 }
44
45 // Pop an element off the stack
46 template< class T >
47 bool Stack< T >::pop( T &popValue )
48 {
49     if ( !isEmpty() ) {
50         popValue = stackPtr[ top-- ]; // remove item from Stack
51         return true; // pop successful
52     }
53     return false; // pop unsuccessful
54 }
55
56 #endif
57 // Fig. 12.3: fig12_03.cpp
58 // Test driver for Stack template
59 #include <iostream.h>
60 #include "tstack1.h"
61
62 int main()
63 {
64     Stack< double > doubleStack( 5 );
65     double f = 1.1;
66     cout << "Pushing elements onto doubleStack\n";
67
68     while ( doubleStack.push( f ) ) { // success true returned
69         cout << f << ' ';
70         f += 1.1;
71     }
72
73     cout << "\nStack is full. Cannot push " << f
74         << "\n\nPopping elements from doubleStack\n";
75
76     while ( doubleStack.pop( f ) ) // success true returned
77         cout << f << ' ';
78
79     cout << "\nStack is empty. Cannot pop\n";
80
81     Stack< int > intStack;
82     int i = 1;
83     cout << "\nPushing elements onto intStack\n";
84
85     while ( intStack.push( i ) ) { // success true returned
86         cout << i << ' ';
87         ++i;
88     }
89
90     cout << "\nStack is full. Cannot push " << i
```

```

91         << "\n\nPopping elements from intStack\n";
92
93     while ( intStack.pop( i ) ) // success true returne
94         cout << i << ' ';
95
96     cout << "\nStack is empty. Cannot pop\n";
97     return 0;
98 }

```

#### 输出结果:

```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements form intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop

```

图 12.3 演示类模板 Stack

下面建立一个测试堆栈类模板（见图 12.5 的输出）的驱动程序（函数 main）。程序在开始的时候实例化了一个大小为 5 的对象 doubleStack。该对象声明为类 Stack< double >（称为 double 类型的 Stack 类）的对象。为了产生出 double 类型的 Stack 类的源代码，编译器会自动把模板中的参数类型 T 替换成 double。尽管程序看不到这个源代码，但仍将其放进源代码中编译。

然后程序成功地把 1.1、2.2、3.3、4.4 和 5.5 这几个 double 值压入（push）堆栈 doubleStack。当试图将第六个值压入堆栈中的时候，push 循环中止（栈已经满了，因为它只能容纳 5 个元素）。

然后程序再将这 5 个元素弹出（pop）堆栈（以 LIFO 顺序）。在试图弹出第六个元素的时，出栈循环中止，因为这时堆栈已经空了。

接下来，程序用下面的声明语句实例化了一个 int 类型的堆栈 intStack：

```
Stack< int > intStack
```

因为没有指定堆栈的大小，所以使用默认构造函数（第 11 行）中的默认值 10 作为堆栈的大小。重复上述操作，用循环结构不断向 intStack 中压入整数值，直到栈满为止，然后再循环从堆栈中弹出数值，直到栈空为止。

在类模板首部以外的成员函数定义都要以下面的形式开头：

```
template < class T >
```

然后，成员函数的定义与普通成员函数的定义相似，只是 Stack 元素的类型要用类型参数 T 表示。二元作用域运算符和 Stack< T > 类模板将成员函数的定义与正确的类模板范围联系起来。本例中，类名是 Stack< T >。当建立类型为 Stack< double > 的对象 doubleStack 的时候，Stack 的构造函数使用 new 建立了一个表示堆栈的 double 类型数组。因此，对于语句：

```
stackPtr = new T [ size];
```

编译器将在模板类 `Stack< double >` 中生成下面的代码:

```
stackPtr new double [ size];
```

注意图 12.3 函数 `main` 中的代码即 `main` 上半部分的 `doubleStack` 操作和 `main` 下半部分的 `intStack` 操作基本相同。这里又可以使用函数模板。图 12.4 的程序用函数模板 `testStack` 进行与图 12.3 相同的工作, 将一系列值压入 `Stack< T >` 中并从 `Stack< T >` 中弹出数值。函数模板 `testStack` 用参数 `T` 表示 `Stack< T >` 中保存的数据类型。该函数模板取 4 个参数: `Stack< T >` 类型对象的引用、类型为 `T` 的值用作压入 `Stack< T >` 的第一个值、类型为 `T` 的值用作压入 `Stack< T >` 的增量值以及 `const char *` 类型的字符串表示输出的 `Stack< T >` 对象名。函数 `main` 只是实例化 `Stack< double >` 类型对象 `doubleStack` 和实例化 `Stack< int >` 类型对象 `intStack`, 如下所示 (第 37 行到第 38 行):

```
testStack( doubleStack, 1.1, 1.1, "doubleStack" );
testStack( intStack, 1, 1, "intStack" );
```

注意图 12.4 的输出与图 12.3 的输出一致。

```
1 // Fig. 12.4: fig12_04.cpp
2 // Test driver for Stack template.
3 // Function main uses a function template to manipulate
4 // objects of type Stack< T >.
5 #include <iostream.h>
6 #include "tstack1.h"
7
8 // Function template to manipulate Stack< T >
9 template< class T >
10 void testStack(
11     Stack< T > &theStack,    // reference to the Stack< T >
12     T value,                // initial value to be pushed
13     T increment,            // increment for subsequent values
14     const char *stackName ) // name of the Stack< T > object
15 {
16     cout << "\nPushing elements onto " << stackName << '\n';
17
18     while ( theStack.push( value ) ) { // success true returned
19         cout << value << ' ';
20         value += increment;
21     }
22
23     cout << "\nStack is full. Cannot push " << value
24         << "\n\nPopping elements from " << stackName << '\n';
25
26     while ( theStack.pop( value ) ) // success true returned
27         cout << value << ' ';
28
29     cout << "\nStack is empty. Cannot pop\n";
30 }
31
32 int main()
33 {
34     Stack< double > doubleStack( 5 );
```

```
35     Stack< int > intStack;
36
37     testStack( doubleStack, 1.1, 1.1, "doubleStack" );
38     testStack( intStack, 1, 1, "intStack" );
39
40     return 0;
41 }
```

**输出结果:**

```
Pushing elements onto doubleStack
1.2 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6
```

```
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop
```

```
Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11
```

```
Popping elements form intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

图 12.4 向函数模板传递 Stack 模板对象

## 12.5 类模板与无类型参数

上节的 Stack 类模板只用模板首部的类型参数,也可以使用无类型参数 (non-type parameter), 无类型参数可以有默认参数,一般将无类型参数当作 const 处理。例如,模板首部可以取 int elements 参数,如下所示:

```
template< class T, int elements > // note non-type parameter
```

然后下列声明:

```
Stack< double, 100 > mostRecentSalesFigures;
```

实例化 (在编译时) 100 个元素的 Stack 模板类 mostRecentSalesFigures (使用 double 值)。这个模板类的类型为 Stack< double, 100 >。类的首部可以包含 private 数据成员, 数组声明如下:

```
T stackHolder[ elements ]; // array to hold stack contents
```

**性能提示 12.2**

如果可能,在编译时指定容器类 (如数组类和堆栈类) 的长度 (可能通过非类型模板长度参数) 可以消除 new 动态生成空间的运行时开销。

**软件工程视点 12.3**

如果可能,在编译时指定容器类的长度 (可能通过非类型模板长度参数) 以避免 new 无法取得所要内存时造成致命的运行时错误。

练习中要用无类型参数生成第8章开发的Array类的模板。这个模板可以用编译时指定类型的指定元素个数实例化Array对象，而不必在运行时动态生成Array对象的空间。

不符合常用类模板的特定类型的类可以重定义该类型的类模板。例如，可以用一个Array类模板实例化任何类型的数组。程序员可以控制某个类型Array类的实例化，如Marian，只要建立类名为Array<Marian>的新类即可。

## 12.6 模板与继承

模板与继承关系如下所示：

- 类模板可以从模板类派生。
- 类模板可以从非模板类派生。
- 模板类可以从类模板派生。
- 非模板类可以从类模板中派生。

## 12.7 模板与友元

函数和整个类都可以声明为非模板类友元。使用类模板，可以声明各种各样的友元关系。友元可以在类模板与全局函数间、另一个类(可能是模板类)的成员函数间或整个类中(可能是模板类)建立。建立这种友元关系的符号可能很繁琐。

在下列X类的类模板中声明为：

```
template< class T > class X
```

下列友元声明：

```
friend void f1();
```

使函数f1成为从上述类模板实例化的每个模板类的友元。

在下列X类的类模板中声明为：

```
template< class T > class X
```

下列友元声明：

```
friend void f2( X< T > &);
```

对特定类型T(如float)使函数f2(X<float>&)成为X<float>的友元。

在类模板中，可以声明另一个类的成员函数是类模板产生的任何模板类的友元。只要用类名和二元作用域运算符指定其它类的成员函数名。例如，在下列X类的类模板中声明为：

```
template< class T > class X
```

下列友元声明：

```
friend void A::f4();
```

使 A 类的成员函数 f4 成为上述类模板实例化的任何模板类的友元。

在下列 X 类的类模板中声明为：

```
template< class T > class X
```

下列友元声明：

```
friend void C< T >::f5( X< T > & );
```

对特定类型 T（如 float）使成员函数：

```
C< float >::f5( X< float > & );
```

成为 X< float >模板类的友元函数。

在下列 X 类的类模板中声明为：

```
template< class T > class X
```

可以声明第二个类 Y，如下所示：

```
friend class Y;
```

使 Y 类的每个成员函数成为 X 的类模板产生的每个模板类的友元。

在下列 X 类的类模板中声明为：

```
template< class T > class X
```

可以声明第二个类 Z，如下所示：

```
friend class Z< T >;
```

使模板类用特定类型 T（如 float）实例化时，class Z< float >的所有成员成为模板类 X< float >的友元。

## 12.8 模板与 static 成员

在非模板类中，类的所有对象共享一个 static 数据成员，static 数据成员应在文件范围内初始化。

从类模板实例化的每个模板类有自己的类模板 static 数据成员，该模板类的所有对象共享一个 static 数据成员。和非模板类的 static 数据成员一样，模板类的 static 数据成员也应在文件范围内初始化。每个模板类有自己的类模板的 static 数据成员副本。

### 小结

- 模板使我们用一个代码段指定一组相关函数（称为模板函数）或一组相关类（称为模板类）。
- 程序员对函数模板的定义只编写一次。基于调用函数时提供的参数类型，C++ 自动产生单独的函数来正确地处理每种类型的调用。这些都是利用程序源代码的剩余空间进行编译。
- 所有函数模板定义都是用关键字 template 开始的，该关键字之后是用尖括号 <> 括起来的形式参数表。函数模板的每个形式类型参数之前应有关键字 class（或新的关键字 typename）。关键字 class 指定函数模板的类型参数，实际上表示“任何内部类型或用户自定义类型”。



- 模板定义的形式参数可用来指定传递给函数的参数类型、函数返回类型和声明函数中变量。
- 形式参数的名字可以只在模板的形式参数表中出现一次。同一个形式参数名可用于多个模板函数。
- 函数模板本身可以用多种方式重载。我们可以提供其他函数模板，指定不同参数的相同函数名。函数模板也可以用其他非模板函数（同名而不同参数）重载。
- 类模板提供了描述一个类和实例化类（即该通用类指定类型的版本）的方法。
- 为了说明如何定制通用类模板以形成指定的模板类，类模板需要类型参数，所以类模板也常常称为参数化类型。
- 要使用模板类的程序员只需简单地编写一个类模板。在需要用模板建立一个新的指定类型的类时，程序员只需要用一种简洁的表示方法，编译器就会写出该模板类的源代码。
- 类模板的定义似乎与普通的类定义没什么不同，除了使用 `template < class T >` 指明这是一个带类型参数 `T`（指明创建的类的类型）的类模板定义。在类首部和成员函数的定义中，类型 `T` 作为一个通用的类型名。
- 在类模板首部以外的成员函数定义都要以 `template < class T >` 开头。接着，成员函数的定义与普通成员函数的定义相似，只是类中的数据通常用类型参数 `T` 表示。二元作用域运算符总是把成员函数的定义与正确的类范围联系起来。
- 类模板首部也可以使用无类型参数。
- 特定类型的类可以重定义该类型的类模板。
- 类模板可以从模板类派生。类模板可以从非模板类派生。模板类可以从类模板派生。非模板类可以从类模板中派生。
- 函数和整个类都可以声明为非模板类的友元。使用类模板，可以声明各种各样的友元关系。友元可以在类模板与全局函数间、另一个类（可能是模板类）的成员函数间或整个类中（可能是模板类）建立。
- 从类模板实例化的每个模板类有自己的类模板的 `static` 数据成员，该模板类的所有对象共享一个 `static` 数据成员。和非模板类的 `static` 数据成员一样，模板类的 `static` 数据成员也应在文件范围内初始化。
- 每个模板类有该类模板的 `static` 数据成员副本。

## 术语

angle brackets 尖括号	keyword template 关键字 template
class template 类模板	non-type parameter in a template header 模板首部中的无类型参数
class template name 类模板名	overloading a template function 重载模板函数
formal parameter in a template header 模板首部中的形式参数	parameterized type 参数化类型
friend of a template 模板的友元	static data member of a class template 类模板的 <code>static</code> 数据成员
function template 函数模板	static data member of a template class 模板类的 <code>static</code> 数据成员
function template declaration 函数模板的声明	static member function of a class template 类模板的 <code>static</code> 成员函数
function template definition 函数模板的定义	
keyword class in a template type parameter 模板类型参数中的关键字 class	

static member function of a template class	模板类	template argument	模板实参
的 static 成员函数		template name	模板名
template class	模板类	template parameter	模板形参
template class member function	模板类成员函数	type parameter in a template header	模板首部的
template< class T >		类型参数	
template function	模板函数	typename	

## 自测练习

12.1 判断下列各题是否正确。如果不正确，请说明原因。

- 函数模板的友元函数必须是模板函数。
- 如果从一个带单个 static 数据成员 的类模板产生几个模板类，则每个模板类共享类模板 static 数据成员的一个副本。
- 模板函数可以用同名的另一模板函数重载。
- 形式参数的名字可以只在模板函数的形式参数表中出现一次。同一个形式参数名只能用于一个模板函数。
- 关键字 class 指定函数模板类型参数，实际上表示“任何用户自定义类型”。

12.2 填空：

- 模板使我们可用一个代码段指定一组相关函数（称为 \_\_\_\_\_）或一组相关类（称 \_\_\_\_\_）。
- 所有的函数模板定义都是以关键字 \_\_\_\_\_ 开始的，该关键字之后是用 \_\_\_\_\_ 括起来的形式参数表。
- 从一个函数模板产生的相关函数都同名，因此编译器用 \_\_\_\_\_ 的解决方法调用相应函数。
- 类模板也称为 \_\_\_\_\_ 类型。
- \_\_\_\_\_ 运算符和模板类名一起将每个成员函数定义与类模板的范围相关联。
- 和非模板类的 static 数据成员一样，模板类的 static 数据成员也应在 \_\_\_\_\_ 范围内初始化。

## 自测练习答案

- 不正确。也可以用非模板函数。
  - 不正确。每个模板类有自己的静态数据成员副本。
  - 正确。
  - 不正确。模板函数间的形式参数名不必惟一。
  - 不正确。这里的关键字 class 也允许内部类型的参数类型。
- 模板函数、模板类。
  - template、尖括号 (<>)。
  - 重载。
  - 参数化。
  - 二元作用域。
  - 文件。

## 练习

- 12.3 根据图 5.15 的排序程序编写函数模板 bubbleSort。编写一个驱动程序，输入、排序和输出 int 数组与 float 数组。
- 12.4 重载图 12.2 的函数模板 printArray，使其取另外两个整数参数 int lowSubscript 和 int highSubscript。调用这个函数只打印数组中的指定部分。验证 lowSubscript 和 highSubscript。如果其中一个值超界或 highSubscript 小于等于 lowSubscript，则重载的 printArray 函数返回 0，否则 printArray 返回打印的元素个数。然后修改 main，对数组 a、b、c 使用两个版本的 printArray。一定要测试两个版本的 printArray 的各种可能情况。
- 12.5 用非模板版本重载图 12.2 的函数模板 printArray，使其以整齐的表格式分栏格式打印字符串数组。
- 12.6 编写判定函数 isEqualTo 的简单函数模板，用相等运算符比较其两个参数，如果相等则返回 1，如果不相等则返回 0。使用这个函数模板，使得程序中对各种内部类型调用 isEqualTo。现在编写程序的另一种形式，对用户自定义类的类型调用 isEqualTo，但不重载相等运算符。运行这个程序时会发生什么情况？重载相等运算符（用运算符函数 operator==），运行这个程序时会发生什么情况？
- 12.7 用无类型参数 numberOfElements 和类型参数 elementType 生成第 8 章开发的 Array 类模板。这个模板按编译时指定个数的指定元素类型实例化 Array 对象。
- 12.8 编写一个使用类模板 Array 的程序，模板可以实例化任何元素类型的 Array 对象。用 float 元素的 Array（class Array< float >）重定义模板。驱动程序演示通过模板实例化 int 类型的 Array，并使用 class Array< float > 中提供的定义实例化 float 类型的 Array。
- 12.9 试区分模板函数与函数模板。
- 12.10 类模板与模板类哪个像是能够绘制形状的模板？为什么？
- 12.11 函数模板与重载有什么关系？
- 12.12 为什么选择函数模板而不选择宏？
- 12.13 使用函数模板与类模板可能造成哪些性能问题？
- 12.14 编译器通过匹配过程确定函数调用时调用哪个模板函数。什么情况下进行匹配会造成编译错误？
- 12.15 为什么类模板也称为参数化类型？
- 12.16 解释 C++ 程序中使用下列语句的原因。

```
Array< Employee > workerList( 100 );
```

- 12.17 分析练习 12.16 的答案，解释 C++ 程序中使用下列语句的原因。

```
Array< Employee > workerList;
```

- 12.18 解释 C++ 程序中使用下列语句的原因。

```
template< class T > Array< T >::Array( int s )
```

- 12.19 为什么数组、堆栈之类的容器类模板通常用无类型参数？
- 12.20 说明如何提供特定类型的类重定义该类型的类模板。

12.21 说明类模板与继承的关系。

12.22 假设类模板的首部如下：

```
template< class T1 > class C1
```

说明类模板首部中用下列友元声明时的友元关系。以f开头的标识符是函数，以C开头的标识符是类，以T开头的标识符是任何类型（即内部类型或类类型）。

- a) friend void f1();
- b) friend void f2( C1< T1 > & );
- c) friend void C2::f4();
- d) friend void C3< T1 >::f5( C1< T1 > & );
- e) friend class C5;
- f) friend class C6< T1 >;

12.23 假设类模板 Employee 有 static 数据成员 count。假设从类模板实例化三个模板类。那么有多少个 static 数据成员？各有什么限制（如果有）？

## 第13章 异常处理

### 教学目标

- 分别用 `try`、`throw` 和 `catch` 监视、表明和处理异常
- 处理未捕获和未预料异常
- 处理 `new` 故障
- 用 `try`、`throw` 和 `catch` 防止内存泄漏
- 了解标准异常层次

### 13.1 简介

本章介绍异常处理 (exception handling)。C++ 的扩展性可能大大增加错误发生的次数和种类。本章介绍的特性可以让程序员编写更清晰、更健全、更具容错性的程序。利用这类技术开发的新系统已经取得成功。我们还将介绍何时不宜使用异常处理。

本章介绍的异常处理样式和细节基于 Andrew Koenig 和 Bjarne Stroustrup 的论文《Exception Handling for C++ (revised)》，发表于 1990 年 4 月在美国旧金山举行的 USENIX C++ 会议上。

错误处理代码的性质与数量随软件系统的应用和是否发布软件产品而不同。商业化产品通常要比自用软件提供更多的错误处理代码。

处理错误的方法多种多样。通常，错误处理代码是在整个系统代码中分布的。代码中可能出错的地方都要进行错误处理。这种方法的好处是程序员阅读代码时能够直接看到错误处理情况，确定是否实现了正确的错误检查。

但这种方法的问题是代码中受到错误处理的“污染”，使应用程序本身的代码更加晦涩难懂，难以看出代码功能是否正确实现。这样就使代码的理解和维护更加困难。

异常的常见例子有 `new` 无法取得所需内存、数组下标超界、运算溢出、除数为 0 和无效函数参数。

C++ 的新异常处理特性使程序员可以删除程序执行“主线条”中的错误处理代码，从而提高程序的可读性和可维护性。

利用 C++ 式的异常处理，可以捕获所有类型的异常、捕获特定类型的所有异常和捕获相关类型的所有异常。这样就可以减少程序未能捕获的错误，使程序更加健壮。异常处理使程序可以捕获和处理错误，而不是任其发生和造成恶果。如果程序员不提供处理致命错误的措施，则程序终止。

异常处理可以处理除数为 0 之类的同步错误 (synchronous error, 在程序执行除法指令时发生)。在程序执行除法之前，异常处理首先检查除数，如果除数为 0，则抛出 (throw) 异常。

异常处理并不处理异步情况，如磁盘 I/O 完成、网络消息到达、鼠标单击等等，这些情况最好用其他方法处理，如中断处理。

异常处理使得系统从导致异常的错误中恢复。恢复过程即执行异常处理器(exception handler)。

异常处理通常用于发现错误的部分与处理错误的部分在不同部分(不同范围)的情况。与用户进行交互式对话的程序不能用异常处理输入错误。

异常处理特别适合程序无法恢复但需要提供有序整理的情况,然后程序可以正常地结束。

#### 编程技巧 13.1

对发生的范围与处理的范围不同的错误使用异常。而对发生的范围与处理的范围相同的错误使用其他方法。

#### 编程技巧 13.2

避免在异常处理中进行错误处理以外的工作,这样可以提高程序的清晰性。

传统程序控制不用异常处理方法还有另一原因。异常处理是用于错误处理的,不是经常的活动,通常在程序准备终止时使用。这样,C++编译器的编写人员不像对正常应用程序代码一样对其实现最优化性能。

#### 性能提示 13.1

尽管异常处理可以进行错误处理以外的工作,但这样会使程序性能下降。

#### 性能提示 13.2

编译器实现异常处理时通常使异常不发生时的异常处理代码开销极小或为0,而发生异常时,则会发生执行开销。异常处理代码的存在无疑会使程序占用更多内存。

#### 软件工程视点 13.1

使用传统控制结构的控制流通常比使用异常更清晰更有效。

#### 常见编程错误 13.1

传统程序控制使用异常处理方法的另一危险是堆栈解退,异常发生之前分配的资源可能无法释放。这个问题可以通过认真编程而避免。

异常处理能提高程序的容错能力。由于编写错误处理代码变得更为轻松,因此程序员更愿意提供错误处理代码,还可以用各种不同方法捕获异常,如根据类型或指定捕获任何类型的异常。

如今编写的大多数程序都只支持单线程执行。Windows NT、OS/2和各种UNIX操作系统越来越重视多线程。本章介绍的方法在多线程程序中同样适用,但我们没有特别介绍多线程程序。

我们将介绍如何处理未捕获异常,考虑未捕获异常如何处理,介绍相关异常如何用公共基础异常类派生的异常类表示。

C++的异常处理特性随ANSI C++标准化的进行而不断发展。标准化在几十、几百人参与的大型软件项目中特别重要,每个人参与系统的不同组件,这些组件要在整个系统中交互,实现正确的功能。

#### 软件工程视点 13.2

异常处理很适合分别开发组件的系统。异常处理使组件组合更容易。每个组件进行自己的异常检测,这与另一范围的异常处理是分开的。

异常处理可以看成另一种从函数返回控制或退出代码块的方法。通常发生异常时,产生异常的函数的调用者、函数调用者的调用者或更深层的调用者处理这个异常。

## 13.2 何时使用异常处理

虽然程序员可以用异常作为程序控制的替代方法,但异常处理应当只用于异常情况,处理程序组件中与这些异常处理没有直接关系的异常,处理函数、库、类等常用软件组件中的异常和组件本身不处理异常的情况,在大型系统中以统一方式处理异常。

### 编程技巧 13.3

对程序本身很容易处理的简单局部错误使用传统错误处理方法而不用异常处理。

### 软件工程视点 13.3

涉及库时,库函数调用者通常用特定错误处理方法处理库函数中产生的异常。库函数很难进行满足用户独特需求的错误处理。因此,异常适合处理库函数产生的错误。

## 13.3 其他错误处理方法

本章之前介绍了其他错误处理方法,现总结如下:

- 用 `assert` 测试编码和设计错误。如果其返回 `false`,则程序终止,应纠正代码。这种方法在调试时很有用处。
- 忽略异常,这不适合公开发布的软件产品和任务关键的专用软件。但自用软件通常可以忽略许多错误。
- 退出程序,使程序无法运行完毕或产生错误结果。实际上,对于许多错误类型,这是个好办法,特别是对于能让程序运行完毕的非致命错误,因为让程序运行完毕很可能使程序员误以为程序工作很顺利。这种方法也不适合任何任务关键的应用程序。资源问题也很重要,如果程序取得资源,则应先正常返回资源之后再终止。

### 常见编程错误 13.2

退出程序会使其他程序无法使用其资源,从而造成资源泄漏。

- 设置一些错误指示符。这里的问题是程序不一定在发生错误的所有地方都检查这些错误指示符。
- 测试错误条件、发出错误消息和调用 `exit`,向程序环境传递相应的错误代码。
- `setjmp` 和 `longjmp`。这个功能通过 `<setjmp.h>` 实现,可以指定从深层嵌套函数立即转入错误处理器。如果没有 `setjmp/longjmp`,则程序要执行几个返回才能从深层嵌套函数退出。这种方法可以转入某个错误处理器。但其在 C++ 中有危险性,因为其解退堆栈而不调用自动对象的析构函数,从而可能造成严重问题。
- 某些特定错误有专门的处理功能。例如, `new` 无法分配内存时,可以用 `new_handler` 函数处理错误。通过提供函数名作为 `set_new_handler` 的参数可以改变这个函数。13.14 节将介绍 `set_new_handler`。

## 13.4 C++ 异常处理基础: try、throw、catch

C++异常处理用于错误检测函数无法处理错误的情况。这种函数抛出异常( throw an exception ),但不能保证有相关的异常处理器。如果有,则异常处理器捕获和处理这个异常。如果没有该类异常相关的异常处理器,则程序终止。

程序员在 try 块中放上出错时产生异常的代码。try 块后面是一个或几个 catch 块。每个 catch 块指定捕获和处理一种异常,而且每个 catch 块包含一个异常处理器。如果异常与 catch 块中的参数类型相符,则执行该 catch 块的代码。如果找不到相应异常处理器,则调用 terminate 函数(默认调用函数 abort)。

抛出异常时,程序控制离开 try 块,从 catch 块中搜索相应异常处理器(稍后将介绍如何形成相应异常处理器。如果 try 块中没有抛出异常,则跳过该块的异常处理器,程序在最后一个 catch 块之后恢复执行。

我们可以对异常指定函数 throw,也可以指定函数不抛出任何异常。

函数的 try 块中抛出异常,或者从 try 块直接或间接调用的函数抛出异常。执行 throw 的点称为抛出点(throw point)。抛出点也指抛出表达式本身。抛出异常之后,控制无法返回抛出点。

发生异常时,可以从异常点向异常处理器传递信息。这些信息是抛出对象的类型或抛出对象中的信息。

抛出的对象通常是字符串(错误消息)或类对象。抛出对象向处理该异常的异常处理器传递信息。

### 软件工程视点 13.4

异常处理的关键是程序或系统中处理异常的部分可以和检测与产生异常的部分分开。

## 13.5 简单异常处理例子:除数为 0

下面考虑一个简单异常处理例子。图 13.1 的程序用 try、throw 和 catch 检测除数为 0 的异常情况,表示并处理除数为 0 的异常。

```
1 // Fig. 13.1: fig13_01.cpp
2 // A simple exception handling example.
3 // Checking for a divide-by-zero exception.
4 #include <iostream.h>
5
6 // Class DivideByZeroException to be used in exception
7 // handling for throwing an exception on a division by zero.
8 class DivideByZeroException {
9 public:
10     DivideByZeroException()
11         : message( "attempted to divide by zero" ) { }
12     const char *what() const { return message; }
13 private:
14     const char *message;
15 };
16
17 // Definition of function quotient. Demonstrates throwing
```



```
18 // an exception when a divide-by-zero exception is encountered.
19 double quotient( int numerator, int denominator )
20 {
21     if ( denominator == 0 )
22         throw DivideByZeroException();
23
24     return static_cast< double > ( numerator ) / denominator;
25 }
26
27 // Driver program
28 int main()
29 {
30     int number1, number2;
31     double result;
32
33     cout << "Enter two integers (end-of-file to end): ";
34
35     while ( cin >> number1 >> number2 ) {
36
37         // the try block wraps the code that may throw an
38         // exception and the code that should not execute
39         // if an exception occurs
40         try {
41             result = quotient( number1, number2 );
42             cout << "The quotient is: " << result << endl;
43         }
44         catch ( DivideByZeroException ex ) { // exception handler
45             cout << "Exception occurred: " << ex.what() << '\n';
46         }
47
48         cout << "\nEnter two integers (end-of-file to end): ";
49     }
50
51     cout << endl;
52     return 0;        // terminate normally
53 }
```

**输出结果:**

```
Enter two integers (end-of-file to end); 100 7
The quotient is: 14.2857
```

```
Enter two integers (end-of-file to end); 100 0
Exception occurred: attempted to divide by zero
```

```
Enter two integers (end-of-file to end); 33 9
The quotient is: 3.66667
```

```
Enter two integers (end-of-file to end):
```

图 13.1 简单的异常处理例子: 除数为0的异常

第一个输出显示执行成功。第二个除数为0, 程序发现错误并发出错误消息。

现在考虑 main 中的驱动程序。程序提示并输入两个整数。注意 number1 和 number2 的局部声明。

然后程序继续执行try块,其中的代码可能抛出异常。注意try块中并没有显式列出可能造成错误的实际除法,而是通过quotient函数调用实际除法的代码。函数quotient实际抛出除数为0的异常对象,将在稍后介绍。一般来说,错误可能通过try块中的代码体现,可能通过函数调用体现,也可能通过try块的代码中启动的深层嵌套函数调用体现。

try块后面是个catch块,包含除数为0的异常的异常处理器。一般来说,try块中抛出异常时,catch块捕获这个异常,表明符合所抛出异常的相应类型。图13.1中,catch块指定捕获DivideByZeroException类型的异常对象,这种对象符合函数quotient中抛出的异常对象。该异常处理器发出一个错误消息并返回,这里返回1表示因为错误而终止。异常处理器也可以更加复杂。

执行时,如果try块中的代码没有抛出异常,则立即跳过try块后面的所有catch异常处理器,执行catch异常处理器后面的第一条语句,图13.1中执行return语句,返回0表示正常终止。

下面看看DivideByZeroException类和quotient函数的定义。在函数quotient中,if语句确定除数为0时,if语句体发出一个throw语句,指定异常对象的构造函数名。这样就生成DivideByZeroException的类对象。try块之后的catch语句(指定类型DivideByZeroException)捕获这个对象。DivideByZeroException类的构造函数只是将message数据成员指向字符串"Divide by zero"。catch处理器指定的参数(这里为参数error)中接收抛出的对象,并通过调用public访问函数printMessage打印这个消息。

#### 编程技巧 13.4

将每种运行时错误与相应命名的异常对象相关联能提高程序的清晰性。

## 13.6 抛出异常

throw关键字表示发生了异常,称为抛出异常。throw通常指定一个操作数(我们将介绍不指定操作数的特殊情况)。throw的操作数可以是任何类型,如果操作数是个对象,则称为异常对象。也可以抛出条件表达式而不是抛出对象,可以抛出不用于错误处理的对象。

抛出异常时,指定相应类型的最近一个异常处理器(对抛出该异常的try块)捕获这个异常。try块的异常处理紧接在try块后面。

抛出异常时,生成和初始化throw操作数的一个临时副本,然后这个临时对象初始化异常处理器中的参数。异常处理器执行完毕和退出时,删除临时对象。

#### 软件工程视点 13.5

如果需要传递导致异常的错误信息,则可以把这种信息放在抛出对象中。catch处理器包含引用该信息的参数名。

#### 软件工程视点 13.6

也可以抛出对象而不传递信息,这时只要知道抛出这种类型的对象提供了异常处理器完成工作所需的足够信息。

抛出异常时,控制退出当前try块,进入try块后面相应的catch处理器(如果有)。抛出点也可能深深嵌套在try块内,控制仍将传入这个catch处理器;抛出点也可能深深嵌套在函数调用中,控制也会转入这个catch处理器。

try 块可能不包含错误检查和 throw 语句, 但 try 块中所指的代码可能导致执行构造函数中的错误检查代码。try 块代码可能对数组类对象加上数组下标, 其 operator[] 成员函数可以重载成抛出下标越界错误的异常。任何函数调用均可调用可能抛出异常的代码或另一可能抛出异常的函数。

尽管异常可以终止程序执行, 但也不一定需要终止程序执行。但异常会终止异常所在的程序块。

#### 常见编程错误 13.3

异常只能在 try 块中抛出, 如果在 try 块之外抛出异常, 则可能调用 terminate。

#### 常见编程错误 13.4

可以抛出条件异常。但一定要小心, 因为上升规则可能使条件表达式返回的值与所要类型不同。例如, 从同一条件表达式抛出 int 或 double 时, 条件表达式将 int 变成 double。因此, 结果总是由参数为 double 的处理器捕获, 而不是有时由参数为 double 的处理器捕获, 有时由参数为 int 的处理器捕获。

## 13.7 捕获异常

异常处理器放在 catch 块中。每个 catch 块以关键字 catch 开始, 接着是括号内包含的类型 (表示该块处理的异常类型) 和可选参数名, 后面是用花括号括起来的描述异常处理器的代码。捕获异常时, 执行 catch 块中的代码。

catch 处理器定义自己的范围。catch 在括号中指定要捕获的对象类型。catch 处理器中的参数可以命名也可以无名。如果是命名参数, 则可以在处理器中引用这个参数。如果是无名参数 (只指定匹配抛出对象类型的类型), 则信息不从抛出点传递到处理器中, 只是把控制从抛出点转到处理器中, 许多异常都可以这样。

#### 常见编程错误 13.5

指定逗号分开的 catch 参数表是个语法错误。

抛出异常对象类型与 catch 处理器参数类型相符时, 执行该类型的 catch 块, 即执行该类型的异常处理器。

catch 处理器中在当前活动 try 块后面第一个符合所抛出对象的处理器捕获异常。稍后将介绍匹配规则。

没有捕获的异常调用 terminate (默认调用 abort) 终止程序。也可以指定自定义行为, 在 set\_terminate 函数调用中指定函数名参数, 从而执行另一个函数。

catch 后面是括号和省略号:

```
catch (...)
```

表示捕获所有异常。

#### 常见编程错误 13.6

将 catch (...) 放在其他 catch 块前面时, 其他块根本无法执行。catch (...) 总是作为 try 块后面的处理器列表中最后一个处理器。

#### 软件工程视点 13.7

用 catch (...) 捕获异常的一个缺点是通常无法确定异常类型。另一个缺点是没有命名参数, 就无法在异常处理器中引用异常对象。

也许某个抛出对象没有任何匹配的异常处理器。这时匹配搜索会继续到外面一层 try 块。这个过程一直继续，也许最终还是没有任何匹配的异常处理器。这时调用 `terminate`（默认调用 `abort`）终止程序。

异常处理器按顺序搜索，寻找匹配项，并执行第一个匹配的处理器。处理器执行完毕时，控制恢复到最后一个 catch 块后面的第一条语句，即该 try 块中最后一个异常处理器后面的第一条语句。

也许几个异常都匹配所抛出的对象。这时执行第一个匹配的异常处理器。如果几个异常处理器都匹配所抛出的对象，而每个异常处理器用不同方法处理异常，则处理器的顺序会影响处理异常的方法。

也许几个 catch 处理器中都包含匹配所抛出对象的类类型。这可能有几个原因：第一，有一个捕获任何异常的 `catch(...)` 处理器。第二，由于继承层次，派生类对象可以由派生类类型的异常处理器和基类类型的异常处理器捕获。

#### 常见编程错误 13.7

将捕获基类类型的异常处理器放在捕获派生类类型的异常处理器之前是个逻辑错误。基类类型的异常处理器捕获从该类派生的所有对象，因此根本不会执行派生类类型的异常处理器。

#### 测试与调试提示 13.1

程序员确定异常处理器列出的顺序。这个顺序可能影响 try 块中所产生异常的处理方法。如果程序处理异常时出现意外行为，可能是前面的 catch 块捕获并处理了这个异常，使其没有被所需的异常处理器处理。

有时程序可能处理许多密切相关的异常类型。这时不是提供不同的异常类和 catch 处理器，而是用一个异常类和 catch 处理器处理一组异常。发生每个异常时，可以生成具有不同 private 数据的异常对象。catch 处理器通过检查 private 数据区分异常的类型。

何时发生匹配呢？下列情况下，catch 处理器参数匹配所抛出对象的类型：

- 实际是同一类型。
- catch 处理器参数类型是所抛出对象类型的 public 基类。
- 处理器参数为基类指针或引用类型，而抛出对象为派生类指针或引用类型。
- catch 处理器为 `catch(...)`。

#### 常见编程错误 13.8

将带 `void *` 参数类型的异常处理器放在具有其他指针类型的异常处理器前面是个逻辑错误。`void *` 处理器捕获所有指针类型的异常，因此其他异常处理器根本不可能执行。

需要有准确的类型匹配。寻找处理器时只允许派生类向基类转换，而不允许其他转换和升级。

可以抛出 `const` 对象。这时 catch 处理器参数类型也应声明为 `const`。

默认情况下，如果异常没有相应的处理器，则程序终止。尽管这应该是正确的做法，但程序员通常不这样做，从而造成错误，使得程序继续执行。

一个 try 块后面跟几个 catch 块类似于 switch 语句。不必用 `break` 退出异常处理器（跳过其余异常处理器）。每个 catch 块定义不同的范围，而 switch 语句中的所有 case 都在 switch 的范围中。

#### 常见编程错误 13.9

将分号放在 try 块后面或 try 块后面的任何 catch 处理器（除最后一个 catch 处理器）后面是个语法错误。

异常处理器无法访问 try 块中定义的自动化对象, 因为发生异常时 try 块终止, try 中的所有自动对象均在处理器开始执行之前删除。

异常处理器中发生异常时会出现什么情况呢? 异常处理器开始执行时, 原先捕获的异常正在处理。因此异常处理器中发生异常时应在抛出原异常的 try 块之外处理。

异常处理器可以用不同的方式编写, 可以检查错误和确定调用 terminate; 可以再抛出异常 (见 13.8 节); 可以将一种异常变为另一种异常, 抛出不同异常; 可以进行必要的恢复, 并恢复执行最后一个异常处理器之后的语句; 可以检查错误原因、删除错误原因和重新调用原先导致异常的函数 (这不会生成无穷递归); 可以向程序环境返回一些状态值等等。

#### 软件工程视点 13.8

最好在设计过程中把异常处理策略加进系统, 后面要再把异常处理策略加进系统是很困难的。

try 块不抛出任何异常而正常地执行完毕时, 控制传入 try 块后面最后一个 catch 处理器之后的第一条语句。

在 catch 处理器中用 return 语句无法返回抛出点。这种 return 语句只是返回调用 catch 块所在函数的函数。

#### 常见编程错误 13.10

如果认为处理异常后控制会返回 throw 后面的第一条语句是个逻辑错误。

#### 软件工程视点 13.9

传统控制流程不用异常的另一个原因是这些“增加的”异常可能搞乱真正的错误型异常。程序员更难跟踪异常种类。例如, 程序处理大量异常时, 无法确定 catch(...) 捕获的是什么异常。异常情况只能针对不常见的极少数情况。

捕获异常时, try 块中可能有已经分配而还没有释放的资源。如果可能, catch 处理器应释放这些资源。例如, catch 处理器删除 new 分配的空间和关闭抛出异常的 try 块中打开的任何文件。

catch 处理器处理错误之后可以让程序继续正确执行, 也可以终止程序。

catch 处理器本身也可以发现错误和抛出异常。这种异常不是由抛出异常的 catch 处理器所在 try 块中的 catch 处理器处理, 而是由外层 try 块中的相关 catch 处理器处理。

#### 常见编程错误 13.11

假设 catch 处理器抛出的异常由该处理器处理, 或由抛出异常的 try 块中相关的处理器 (导致执行原先的 catch 处理器) 处理是个逻辑错误。

## 13.8 再抛出异常

捕获异常的处理器也可以决定不处理异常或释放资源, 然后让其他处理器处理这个异常。这时, 处理器只要再抛出异常, 如下所示:

```
throw;
```

这种不带参数的 throw 再抛出异常。如果开始没有抛出异常, 则再抛出异常调用 terminate。

#### 常见编程错误 13.12

将空 throw 语句放在 catch 处理器之外, 执行这种 throw 会调用 terminate。

即使处理器能处理异常,不管这个异常是否进行处理,处理器仍然可以再抛出异常以便在处理器之外继续处理。再抛出异常由外层 try 块检测,由所在 try 块之后列出的异常处理器处理。

#### 软件工程视点 13.10

用 catch(...)进行与异常类型无关的恢复,如释放共用资源。可以再抛出异常以警示更具体的外层 catch 块。

图 13.2 的程序演示了再抛出异常。在 main 第 26 行的 try 块中,调用函数 throwException。在函数 throwException 的 try 块中,第 13 行的 throw 语句抛出标准库类 exception 的实例(在头文件 <exception> 中定义)。这个异常在第 15 行的 catch 处理器中立即捕获,打印一个错误消息,然后再抛出异常。因此终止函数 throwException 并将控制返回 main 中的 try/catch 块。第 30 行再次捕获异常并打印一个错误消息。

```
1 // Fig. 13.2: fig13_02.cpp
2 // Demonstration of rethrowing an exception.
3 #include <iostream>
4 #include <exception>
5
6 using namespace std;
7
8 void throwException() throw ( exception )
9 {
10     // Throw an exception and immediately catch it.
11     try {
12         cout << "Function throwException\n";
13         throw exception(); // generate exception
14     }
15     catch( exception e )
16     {
17         cout << "Exception handled in function throwException\n";
18         throw; // rethrow exception for further processing
19     }
20
21     cout << "This also should not print\n";
22 }
23
24 int main()
25 {
26     try {
27         throwException();
28         cout << "This should not print\n";
29     }
30     catch ( exception e )
31     {
32         cout << "Exception handled in main\n";
33     }
34
35     cout << "Program control continues after catch in main"
36         << endl;
37     return 0;
38 }
```

#### 输出结果:

Function throwException

```
Exception handled in function throwException
Exception handled in main
Program control continues after catch in main
```

图 13.2 再抛出异常

## 13.9 异常指定

异常指定可以由指定函数抛出一列异常：

```
int g( float h ) throw (a, b, c)
{
    // function body
}
```

可以限制从函数抛出的异常类型。函数声明中可以指定异常类型作为异常指定（也称为抛出表，throw list）。异常指定列出可抛出的异常。函数可以抛出指定异常或派生类型。尽管这样好像保证不会抛出其他异常类型，但其实也可以抛出其他异常类型。如果抛出异常指定中没有列出的异常，则调用函数 `unexpected`。

将 `throw()`（即空异常指定）放在函数的参数表之后表示该函数不抛出任何异常。但这种函数仍然可以抛出异常，这时也调用函数 `unexpected`。

### 常见编程错误 13.13

如果抛出函数异常指定中没有的异常，则调用函数 `unexpected`。

不带异常指定的函数可以抛出任何异常：

```
void g(); // this function can throw any exception
```

`unexpected` 函数的含义可以调用函数 `set_unexpected` 重新定义。

异常处理的一个有趣方面是函数在 `throw` 表达式中包含函数异常指定中未列出的异常时，编译器不产生语法错误。函数在执行时抛出这个异常之后才会捕获错误。

如果函数抛出特定类的类型的异常，则函数也可以抛出用 `public` 继承从该类派生的所有类异常。

## 13.10 处理意外异常

函数 `unexpected` 调用 `set_unexpected` 函数指定的函数。如果没有用 `set_unexpected` 函数指定函数，则默认调用 `terminate`。

函数 `terminate` 可以显式调用，在无法捕获抛出的异常时、在异常处理期间打乱堆栈时、作为调用 `unexpected` 的默认操作时或在异常导致堆栈解退时析构函数抛出异常的情况下都会调用 `terminate`。

函数 `set_terminate` 可以指定调用 `terminate` 时调用的函数，否则 `terminate` 调用 `abort`。

函数 `set_terminate` 和 `set_unexpected` 的函数原型分别在头文件 `<terminate.h>` 和 `<unexpected.h>` 中。

函数 `set_terminate` 和 `set_unexpected` 分别返回 `terminate` 和 `unexpected` 调用的最后一个函数的指针。这样就使程序员可以保存函数指针，以便后面恢复。

函数 `set_terminate` 和 `set_unexpected` 取函数指针为参数。每个参数指向返回类型为 `void` 和无参数的函数。

如果用户自定义终止函数的最后一个操作不是退出程序,则执行用户自定义终止函数的其他语句之后自动调用 `abort` 函数终止程序。

## 13.11 堆栈解退

特定范围中抛出异常而未捕获时,函数调用堆栈解退,试图在外层 `try/catch` 块中捕获这个异常。函数调用堆栈解退表示未捕获异常的函数终止,删除函数的所有局部变量,控制返回函数调用点。如果函数调用点在 `try` 块中,则试捕获这个异常,如果函数调用点不在 `try` 块中或没有捕获这个异常,则再次发生堆栈解退。上节曾介绍过,如果程序不捕获异常,则调用函数 `terminate` 以终止程序。图 13.3 的程序演示了堆栈解退。

```
1 // Fig. 13.3: fig13_03.cpp
2 // Demonstrating stack unwinding.
3 #include <iostream>
4 #include <stdexcept>
5
6 using namespace std;
7
8 void function3() throw ( runtime_error )
9 {
10     throw runtime_error( "runtime_error in function3" );
11 }
12
13 void function2() throw ( runtime_error )
14 {
15     function3();
16 }
17
18 void function1() throw ( runtime_error )
19 {
20     function2();
21 }
22
23 int main()
24 {
25     try {
26         function1();
27     }
28     catch ( runtime_error e )
29     {
30         cout << "Exception occurred: " << e.what() << endl;
31     }
32
33     return 0;
34 }
```

**输出结果:**

Exception occurred: runtime\_error in function3

图 13.3 堆栈解退



在 `main` 中, 第 25 行的 `try` 块调用 `function1`。然后第 18 行定义的 `function1` 调用 `function2`。然后第 13 行定义的 `function2` 调用 `function3`。`function3` 的第 10 行抛出 `exception` 对象。由于第 10 行不在 `try` 块中, 因此发生堆栈解退, `function3` 终止, 控制返回 `function2` (第 15 行)。由于第 15 行不在 `try` 块中, 再次发生堆栈解退, `function2` 终止, 控制返回 `function1` (第 20 行)。由于第 20 行不在 `try` 块中, 又一次发生堆栈解退, `function1` 终止, 控制返回 `main` (第 26 行)。由于第 26 行在 `try` 块中, 可以捕获异常, 并在 `try` 块后面第一个匹配的 `catch` 处理器中处理 (第 28 行)。

## 13.12 构造函数、析构函数与异常处理

首先要处理前面已经提到但还没有完全解决的问题。构造函数中发现错误时会发生什么情况? 例如, `String` 构造函数在 `new` 失败和无法取得保持 `String` 的内部表示所需空间时如何响应? 问题是构造函数无法返回数值, 如何让外部知道对象没有顺利构造呢? 一种方案是返回没有正确构造的对象, 希望对象使用者通过相应测试确定该对象是不能使用的对象。另一种方案是在构造函数之外设置一些变量。抛出的异常向外部传递失败的构造函数信息, 并负责处理这个故障。

要捕获异常, 异常处理器要访问所抛出对象的复制构造函数 (默认成员的副本也有效)。

构造函数中抛出异常时, 对抛出异常之前要构造的对象调用析构函数。

抛出异常之前每个 `try` 块中构造的自动对象都调用析构函数。异常在开始执行处理器时处理, 这时堆栈解退一定已经完成。如果堆栈解退调用析构函数而抛出异常, 则调用 `terminate`。

如果对象有成员函数, 且如果异常在外层对象构造完成之前抛出, 则执行发生异常之前所构造成员对象的析构函数。如果发生异常时部分构造了对象数组, 则只调用已构造数组元素的析构函数。

异常可能越过通常释放资源的代码, 从而造成资源泄漏。要解决这个问题, 一种方法是在请求资源时初始化一个局部对象, 发生异常时, 调用析构函数并释放资源。

要捕获析构函数中抛出的异常, 可以将调用析构函数的函数放在 `try` 块中, 并提供相应类型的 `catch` 处理器。所抛出对象的析构函数在异常处理器执行完毕之后执行。

## 13.13 异常与继承

可以从共用基类派生各种异常类。如果 `catch` 捕获基类类型异常对象的指针或引用, 则也可以捕获该基类所派生的异常对象的指针或引用。这样允许相关错误的多态处理。

### 测试与调试提示 13.2

利用异常继承使异常处理器可以用相当简单的符号捕获相关错误。虽然可以捕获每个派生类异常对象的指针或引用, 但更简练的方法是捕获基类异常对象的指针或引用。另外, 如果程序员忘记测试一个或几个派生类指针或引用, 则捕获每个派生类异常对象的指针或引用容易造成错误。

## 13.14 处理 `new` 故障

处理 `new` 故障的方法有多种。到目前为止, 我们介绍过用宏 `assert` 测试 `new` 返回的值。如果返回值为 0, 则 `assert` 宏终止程序。这不是处理 `new` 故障的健壮机制, 它不允许我们用任何方法从故障恢复。ANSI/ISO C++ 草案标准指定, 出现 `new` 故障时抛出 `bad_alloc` 异常 (在头文件 `<new>` 中定

义)。但许多编译器目前还不支持草案标准，仍然在 new 故障时返回 0。本节介绍三个 new 故障的例子。第一个例子用 Microsoft 的 Visual C++ 5.0 编译，仍然在 new 故障时返回 0。第二和第三个例子用 Metrowerks Code Warrior Professional Release 1 中的 C++ 编译器编译，出现 new 故障时，它抛出 bad\_alloc 异常。我们在不同的运行 Windows 95 的计算机上测试三个程序，每台机器的内存和硬盘空间各不相同。

图 13.4 演示了 new 请求分配内存失败时返回 0。第 10 行的 for 结构循环 10 次，每次循环分配 5 000 000 个 double 值的数组（即 40 000 000 字节，因为 double 通常为 8 个字节）。第 13 行的 if 结构测试每次 new 操作中是否顺利分配内存。如果 new 请求内存失败并返回 0，则打印 "Memory allocation failed\n" 消息，循环终止。

```
1 // Fig. 13.4: fig13_04.cpp
2 // Demonstrating new returning 0
3 // when memory is not allocated
4 #include <iostream.h>
5
6 int main()
7 {
8     double *ptr[ 10 ];
9
10    for ( int i = 0; i < 10; i++ ) {
11        ptr[ i ] = new double[ 5000000 ];
12
13        if ( ptr[ i ] == 0 ) { // new failed to allocate memory
14            cout << "Memory allocation failed for ptr[ "
15                << i << " ]\n";
16            break;
17        }
18        else
19            cout << "Allocated 5000000 doubles in ptr[ "
20                << i << " ]\n";
21    }
22
23    return 0;
24 }
```

**输出结果：**

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Memory allocation failed for ptr[ 2 ]
```

图 13.4 new 请求内存失败时返回 0

输出表示 new 失败和循环终止之前只进行了两次循环。不同系统中的输出可能不同，取决于实际内存和可用的虚拟内存的磁盘空间。

图 13.5 演示了 new 请求内存失败时返回 bad\_alloc。第 12 行的 for 结构循环 10 次，每次循环分配 5 000 000 个 double 值的数组（即 40 000 000 字节，因为 double 通常为 8 个字节）。如果 new 失败并抛出 bad\_alloc 异常，则循环终止，程序继续第 18 行的异常处理流程捕获和处理异常，打印 "Exception occurred:" 消息，然后 exception.what() 返回异常特定消息的字符串（对于 bad\_alloc 为 "Allo-

cation Failure")。输出表示 new 失败和抛出 bad\_alloc 异常之前只进行了三次循环。不同系统中的输出可能不同,取决于实际内存和可用的虚拟内存的磁盘空间。

```
1 // Fig. 13.5: fig13_05.cpp
2 // Demonstrating new throwing bad_alloc
3 // when memory is not allocated
4 #include <iostream>
5 #include <new>
6
7 int main()
8 {
9     double *ptr[ 10 ];
10
11     try {
12         for ( int i = 0; i < 10; i++ ) {
13             ptr[ i ] = new double[ 5000000 ];
14             cout << "Allocated 5000000 doubles in ptr[ "
15                 << i << " ]\n";
16         }
17     }
18     catch ( bad_alloc exception ) {
19         cout << "Exception occurred: "
20             << exception.what() << endl;
21     }
22
23     return 0;
24 }
```

**输出结果:**

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Exception occurred: Allocation Failure
```

图 13.5 new 请求内存失败时返回 bad\_alloc

编译器对 new 故障处理的方法各不相同。一般情况下,许多当前的和原来的 C++ 编译器在 new 失败时默认返回 0。有些编译器在包括头文件 <new> (或 <new.h>) 时支持抛出异常。有些编译器 (如 CodeWarrior Professional Release 1) 不管是否包括头文件 <new> 默认抛出 bad\_alloc。详见编译器文档中关于编译器对 new 故障处理方法的说明。

ANSI/ISO C++ 草案标准指定标准支持的编译器在 new 失败时仍然可以用返回 0 的版本。为此,头文件 <new> 定义类型 nothrow, 使用如下:

```
double *ptr = new( nothrow ) double [ 5000000 ];
```

上述语句表示用不支持抛出 bad\_alloc 异常的 new 版本 (即 nothrow) 分配 5 000 000 个 double 值的数组。

#### 软件工程视点 13.11

为了使程序更健壮, ANSI/ISO C++ 草案标准建议程序员应使用抛出 bad\_alloc 异常的 new 版本。

还可以用其他特性进行 new 故障处理。函数 set\_new\_handler (原型在头文件 <new> 或 <new.h> 中) 取一个函数指针作为参数, 所指函数不取参数并返回 void。函数指针注册为 new 失败时要调用

的函数。这样就向程序员提供了处理每个new故障的一致方法,而不管故障发生在程序中哪个地方。程序中用set\_new\_handler注册new处理器之后,new不会在故障时抛出bad\_alloc。

new运算符实际上是一个循环,请求所要的内存。如果内存分配成功,则new返回该内存的指针。如果内存分配失败,且没有用set\_new\_handler注册new处理器函数,则new抛出bad\_alloc异常。如果new无法分配内存而注册了new处理器函数,则调用new处理器函数。C++草案标准指定new处理器函数应完成下列任务:

1. 通过删除其他动态分配内存以获得更多的内存空间,然后返回new运算符中的循环,试图再次分配内存。
2. 抛出bad\_alloc类型的异常。
3. 调用函数abort或exit(都在<csdlib>或<stdlib.h>头文件中定义)终止程序。

图13.6的程序演示set\_new\_handler。函数customNewHandler只是打印错误消息和调用abort终止程序。输出显示new失败和抛出bad\_alloc异常之前循环进行了三次迭代。不同系统中的输出可能不同,取决于实际内存和可用的虚拟内存的磁盘空间。

```
1 // Fig. 13.6: fig13_06.cpp
2 // Demonstrating set_new_handler
3 #include <iostream.h>
4 #include <new.h>
5 #include <stdlib.h>
6
7 void customNewHandler()
8 {
9     cerr << "customNewHandler was called";
10    abort();
11 }
12
13 int main()
14 {
15     double *ptr[ 10 ];
16     set_new_handler( customNewHandler );
17
18     for ( int i = 0; i < 10; i++ ) {
19         ptr[ i ] = new double[ 5000000 ];
20
21         cout << "Allocated 5000000 doubles in ptr[ "
22              << i << " ]\n";
23     }
24
25     return 0;
26 }
```

**输出结果:**

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
CustomNewHandler was called
```

图13.6 演示set\_new\_handler

## 13.15 auto\_ptr 类与动态内存分配

一个常见的编程习惯是在空闲存储区中动态分配内存(可能是对象),将该内存的地址赋给一个指针,使用这个指针操作内存,并在该内存不再需要时用delete释放内存。如果内存分配之后和执行delete语句之前发生异常,则可能发生内存泄漏。ANSI/ISO C++草案标准提供<memory>头文件中的auto\_ptr类模板,可以解决这个问题。

auto\_ptr类对象维护动态分配内存的指针。当auto\_ptr对象超出范围时,对指针数据成员进行一个delete操作。auto\_ptr类模板提供了\*和->运算符,因此auto\_ptr对象可以像普通指针变量一样使用。图13.7演示auto\_ptr对象指向Integer类对象(在第8行到第18行定义)。

```
1 // Fig. 13.7: fig13_07.cpp
2 // Demonstrating auto_ptr
3 #include <iostream>
4 #include <memory>
5
6 using namespace std;
7
8 class Integer {
9 public:
10     Integer( int i = 0 ) : value( i )
11     { cout << "Constructor for Integer " << value << endl; }
12     ~Integer()
13     { cout << "Destructor for Integer " << value << endl; }
14     void setInteger( int i ) { value = i; }
15     int getInteger() const { return value; }
16 private:
17     int value;
18 };
19
20 int main()
21 {
22     cout << "Creating an auto_ptr object that points "
23         << "to an Integer\n";
24
25     auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
26
27     cout << "Using the auto_ptr to manipulate the Integer\n";
28     ptrToInteger->setInteger( 99 );
29     cout << "Integer after setInteger: "
30         << ( *ptrToInteger ).getInteger()
31         << "\nTerminating program" << endl;
32
33     return 0;
34 }
```

### 输出结果:

```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7
Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99
Terminating program
```

```
Destructor for Integer 99
```

图 13.7 演示 auto\_ptr

第 25 行:

```
auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
```

生成 auto\_ptr 对象 ptrToInteger 并用动态分配 Integer 对象的指针 ( 包含数值 7 ) 初始化。

第 28 行:

```
ptrToInteger->setInteger( 99 );
```

用 auto\_ptr 重载 -> 运算符和函数调用运算符()调用 ptrToInteger 所指 Integer 对象的 setInteger 函数。

第 30 行:

```
( *ptrToInteger ).getInteger()
```

用 auto\_ptr 重载 \* 运算符复引用 ptrToInteger, 然后用圆点运算符和函数调用运算符()调用 ptrToInteger 所指 Integer 对象的 getInteger 函数。

由于 ptrToInteger 是 main 中的局部自动变量, 因此 main 终止时删除 ptrToInteger。这样就强制删除 ptrToInteger 所指的 Integer 对象, 从而强制调用 Integer 类的析构函数。更重要的是这个方法可以防止内存泄漏。

## 13.16 标准库异常层次

经验表明, 异常是可以分类的。C++ 草案标准提供了标准库异常层次。这个层次以基类 exception 开始 ( 在头文件 <exception> 中定义 ), 该基类提供服务 what(), 在每个派生类中重定义, 发出相应的错误消息。

从基类 exception 可以派生直接派生类 runtime\_error 和 logic\_error ( 都在头文件 <stdexcept> 中定义 ), 每个派生类又可以派生其他类。

从 exception 中还可以派生由于 C++ 语言特性而抛出的异常, 例如, new 抛出 bad\_alloc ( 13.14 节 ), dynamic\_cast 抛出 bad\_cast ( 第 21 章 ), typeid 抛出 bad\_typeid ( 第 21 章 )。如果发生意外异常时, 通过在函数的抛出表中加上 std::bad\_exception, unexpected() 抛出 bad\_exception 而不是 ( 默认 ) 终止程序或调用 set\_unexpected 指定的另一函数。

logic\_error 类是几个标准异常类的基类, 表示程序逻辑中的错误, 可以通过编写正确的代码来防止。下面介绍其中的一些类。invalid\_argument 类表示向函数传入无效参数 ( 可以通过编写正确的代码来防止 )。length\_error 类表示长度大于所操作对象允许的最大长度 ( 第 19 章处理 string 时会抛出 length\_error 异常 )。out\_of\_range 类表示数组和 string 下标之类的值超界。

runtime\_error 类是几个其他异常类的基类, 表示程序中只能在执行时发现的错误。overflow\_error 类表示发生运算上溢错误; underflow\_error 类表示发生运算下溢错误。

### 软件工程视点 13.12

标准 exception 层次只是一个起点, 用户可以抛出标准异常、抛出从标准异常派生的异常或抛出不是从标准异常派生的异常。

#### 常见编程错误 13.14

用户自定义异常类不一定从 `exception` 类派生。因此 `catch(exception e)` 并不保证捕获程序可能遇到的各种异常。

#### 测试与调试提示 13.3

要捕获 `try` 块可能抛出的所有异常，用 `catch(...)`。

### 小结

- 异常的常见例子有 `new` 无法取得所需内存、数组下标超界、运算溢出、除数为0和无效函数参数。
- 异常处理使程序可以捕获和处理错误，而不是任其发生和造成恶果。如果程序员不提供处理致命错误的措施，则程序终止。非致命错误通常允许程序继续执行，但会产生错误结果。
- 异常处理可以处理同步错误作为程序执行的结果。
- 异常处理并不处理异步情况，如磁盘 I/O 完成、网络消息到达、鼠标单击等等，这些情况最好用其他方法处理，如中断处理。
- 异常处理通常用于发现错误的部分与处理错误的部分在不同部分（不同范围）的情况。
- 异常不应为具体的控制流作为一种变换机制使用。控制流和常用的控制结构一般都比异常更清晰、更高效。
- 异常处理应当用于处理程序组件中与这些异常处理没有直接关系的异常。
- 异常处理应当用于处理函数、库、类等常用软件组件中的异常和组件本身不处理异常的情况。
- 异常处理应当用于在大型项目中以统一方式处理整个项目异常。
- C++ 异常处理用于错误检测函数无法处理错误的情况，这种函数抛出异常。如果异常与 `catch` 块中的参数类型相符，则执行该 `catch` 块的代码。如果找不到相应的异常处理器，则调用 `terminate` 函数（默认调用函数 `abort`）。
- 程序员在 `try` 块中放上出错时产生异常的代码。`try` 块后面是一个或几个 `catch` 块。每个 `catch` 块指定捕获和处理一种异常。每个 `catch` 块包含一个异常处理器。
- 抛出异常时，程序控制离开 `try` 块，从 `catch` 块中搜索相应的异常处理器。如果 `try` 块中没有抛出异常，则跳过该块的异常处理器，程序在最后一个 `catch` 块之后恢复执行。
- 函数的 `try` 块中抛出异常，或者从 `try` 块直接或间接调用的函数抛出异常。
- 抛出异常之后，控制无法返回抛出点。
- 发生异常时，可以从异常点向异常处理器传递信息。这些信息是抛出对象的类型或抛出对象中的信息。
- 常见异常类型是 `char *`，该类型只是包括一个错误消息，作为 `throw` 的操作数。
- 异常指定可以由指定函数抛出一列异常，将空异常指定语句放在函数的参数表之后表示该函数不抛出任何异常。
- 抛出异常时，指定相应类型的最近一个异常处理器（对抛出该异常的 `try` 块）捕获这个异常。
- 抛出异常时，生成和初始化 `throw` 操作数的一个临时副本，然后这个临时对象初始化异常处理器中的参数。异常处理器执行完毕和退出时，删除临时对象。
- 不一定总是显式检查错误。`try` 块可能不包含错误检查和 `throw` 语句，但 `try` 块中所指的代码可能导致执行构造函数中的错误检查代码。

- 异常会终止异常所在的程序块。
- 异常处理器放在 catch 块中。每个 catch 块以关键字 catch 开始,接着是括号内的包含类型(表示该块处理的异常类型)和可选参数名。后面是用花括号括起来的描述异常处理器的代码。捕获异常时,执行 catch 块中的代码。
- catch 处理器定义自己的范围。
- catch 处理器中的参数可以命名也可以无名。如果是命名参数,则可以在处理器中引用这个参数,如果是无名参数(只指定匹配抛出对象类型的类型或用省略号表示所有类型),则处理器忽略所有抛出异常。处理器可以将对象重新抛出到外层 try 块中。
- 也可以指定自定义行为,在 set\_terminate 函数调用中指定函数名参数,指定执行另一个函数,代替函数 terminate。
- catch(...)表示捕获所有异常。
- 也许某个抛出对象没有任何匹配的异常处理器。这时匹配搜索会继续到外面一层 try 块。
- 异常处理器按顺序搜索,寻找匹配项,并执行第一个匹配的处理器。处理器执行完毕时,控制恢复到最后一个 catch 块后面的第一条语句。
- 处理器的顺序会影响处理异常的方法。
- 派生类对象可以由派生类类型的异常处理器和基类类型的异常处理器捕获。
- 有时程序可能处理许多密切相关的异常类型。这时不是提供不同的异常类和 catch 处理器,而是可以用一个异常类和 catch 处理器处理一组异常。发生每个异常时,可以生成具有不同 private 数据的异常对象。catch 处理器通过检查 private 数据区分异常的类型。
- 即使有准确匹配也还在匹配时要求标准转换,因为这个处理器出现在导致准确匹配的处理器之前。
- 默认情况下,如果找不到一个异常的处理器,则程序终止。
- 异常处理器无法直接访问 try 块范围中的变量。处理器所需的信息通常在抛出对象中传递。
- 异常处理器可以用不同方式编写,可以检查错误和确定调用 terminate;可以再抛出;通过抛出不同异常可以将一种异常变为另一种异常;可以进行必要的恢复,并恢复执行最后一个异常处理器之后的语句;可以检查错误原因,删除错误原因和重新调用原先导致异常的函数(这不会生成无穷递归);可以向运行环境返回一些状态值等等。
- 应当将捕获基类类型的异常处理器放在捕获派生类类型的异常处理器之后,否则基类类型的异常处理器捕获基类对象和从该类派生的所有对象。
- 捕获异常时,try 块中可能有已经分配而还没有释放的资源。catch 处理器应释放这些资源。
- 捕获异常的处理器也可以决定不处理异常。这时,处理器只要再抛出该异常。这种不带参数的 throw 再抛出异常。如果开始没有抛出异常,则再抛出异常调用 terminate。
- 即使处理器能处理异常,不管这个异常是否进行处理,处理器仍然可以再抛出异常以便在处理器之外继续处理。再抛出异常由外层 try 块检测,由所在 try 块之后列出的异常处理器处理。
- 不带异常指定的函数可以抛出任何异常。
- 函数 unexpected 调用 set\_unexpected 函数指定的函数。如果没有用 set\_unexpected 函数指定函数,则默认调用 terminate。
- 函数 terminate 可以显式调用,也可以在无法捕获抛出的异常时、在异常处理期间打乱堆栈时、作为调用 unexpected 的默认操作时或在异常导致堆栈解退时析构函数抛出异常的情况下调用 terminate。
- 函数 set\_terminate 和 set\_unexpected 的原型分别在头文件 <terminate.h> 和 <unexpected.h> 中。



- 函数 `set_terminate` 和 `set_unexpected` 分别返回 `terminate` 和 `unexpected` 调用的最后一个函数的指针。这样就使程序员可以保存函数指针, 以便后面恢复。
- 函数 `set_terminate` 和 `set_unexpected` 取函数指针为参数。每个参数指向返回类型为 `void` 和无参数的函数。
- 如果用户自定义终止函数的最后一个操作不是退出程序, 则执行用户自定义终止函数的其他语句之后自动调用 `abort` 函数终止程序。
- `try` 块之外抛出的异常会使程序终止。
- 如果 `try` 块后面找不到处理器, 则继续堆栈解退, 直到找到相应处理器。如果最终找不到处理器, 则调用 `terminate` (默认用 `abort`) 退出程序。
- 异常指定列出可抛出的异常。函数可以抛出指定异常或派生类型。如果抛出异常指定中没有指定的异常, 则调用函数 `unexpected`。
- 如果函数抛出特定类类型的异常, 则函数也可以抛出用 `public` 继承从该类派生的所有类异常。
- 要捕获异常, 异常处理器要访问所抛出对象的复制构造函数。
- 构造函数中抛出异常时, 对所有已构造的基类对象和抛出异常之前构造的成员对象调用析构函数。
- 如果发生异常时部分构造了对象数组, 则只调用已构造数组元素的析构函数。
- 要捕获析构函数中抛出的异常, 可以将调用析构函数的函数放在 `try` 块中, 并提供相应类型的 `catch` 处理器。
- 利用异常继承使异常处理器可以用相当简单的符号捕获相关错误。虽然可以捕获每个派生类的异常对象, 但更简练的方法是捕获基类的异常对象。
- ANSI/ISO C++ 草案标准指定, 出现 `new` 故障时抛出 `bad_alloc` 异常 (在头文件 `<new>` 中定义)。
- 许多编译器目前还不支持草案标准, 仍然在 `new` 故障时返回 0。
- 函数 `set_new_handler` (原型在头文件 `<new>` 或 `<new.h>` 中) 取一个函数指针参数, 所指函数不取参数并返回 `void`。函数指针注册为 `new` 失败时要调用的函数。用 `set_new_handler` 注册 `new` 处理器之后, `new` 不会在发生故障时抛出 `bad_alloc`。
- `auto_ptr` 类对象维护动态分配内存的指针。当 `auto_ptr` 对象超出范围时, 对指针数据成员进行一个 `delete` 操作, `auto_ptr` 类模板提供了 `*` 和 `->` 运算符, 因此 `auto_ptr` 对象可以像普通指针变量一样使用。
- C++ 草案标准提供了标准库异常层次。这个层次以基类 `exception` 开始 (在头文件 `<exception>` 中定义), 该基类提供服务 `what()`, 在每个派生类中重定义, 发出相应错误消息。
- 如果发生意外异常时, 通过在函数的抛出表中加上 `std::bad_exception`, `unexpected()` 抛出 `bad_exception` 而不是 (默认) 终止程序或调用 `set_unexpected` 指定的另一函数。

## 术语

<code>abort()</code>	catch an exception 捕获一个异常
<code>assert macro</code> <code>assert</code> 宏	catch argument catch 参数
<code>auto_ptr</code>	catch block catch 块
<code>bad_alloc</code>	catch(...) 空异常指定
<code>bad_cast</code>	dynamic_cast 空 throw 指定
<code>bad_typeid</code>	
catch a group of exceptions 捕获一组异常	

enclosing try block	所在 try 块	nothrow	
exception	异常	out_of_range	
exception declaration	异常声明	overflow_error	
exception handler	异常处理器	rethrow an exception	再抛出异常
<exception> header file	<exception>头文件	robustness	健壮性
exception list	异常表	runtime_error	
exception object	异常对象	set_new_handler()	
exception specification	异常指定	set_terminate()	
exceptional condition	异常条件	set_unexpected()	
exit()		stack unwinding	堆栈解退
fault tolerance	容错	std::bad_exception	
function with no exception specification	不带异常指定的函数	<stdexcept> header file	<stdexcept>头文件
handle an exception	处理异常	terminate()	
handler for a base class	基类处理器	throw an exception	抛出异常
handler for a derived class	派生类处理器	throw an unexpected exception	抛出意外异常
invalid_argument		throw expression	throw 表达式
length_error		throw list	throw 表
logic_error		throw point	throw 点
<memory> header file	<memory>头文件	throw without arguments	不带参数抛出
mission-critical application	任务关键的应用	throw()	
nested exception handlers	嵌套的异常处理器	try block	try 块
new_handler		underflow_error	
<new> header file	<new>头文件	unexpected()	
		uncaught exception	未捕获异常

## 自测练习

- 13.1 列出五个常见的异常例子。
- 13.2 说明异常处理方法不能用于传统程序控制的原因。
- 13.3 为什么异常适合处理库函数产生的错误?
- 13.4 什么是“资源泄漏”?
- 13.5 如果 try 块中不抛出异常, try 块执行完毕之后控制转到哪里?
- 13.6 如果在 try 块之外抛出异常, 会发生什么情况?
- 13.7 说明使用 catch(...)的主要优点和主要缺点。
- 13.8 如果没有匹配所抛出对象类型的 catch 处理器, 会发生什么情况?
- 13.9 如果有多个匹配所抛出对象类型的 catch 处理器, 会发生什么情况?
- 13.10 为什么程序员要指定基类类型为 catch 处理器类型, 然后抛出派生类类型的对象?
- 13.11 catch 处理器如何编写成处理相关错误类型而不用异常类之间的继承?
- 13.12 catch 处理器中用哪种指针类型捕获任何指针类型的所有异常?
- 13.13 假设有准确匹配异常对象类型的 catch 处理器, 什么情况下该异常对象类型会执行不同的 catch 处理器?

- 13.14 抛出异常是否一定终止程序?
- 13.15 `catch` 处理器抛出异常时会发生什么情况?
- 13.16 `throw`; 语句有什么用?
- 13.17 程序员如何限制函数可以抛出的异常类型?
- 13.18 如果函数抛出函数异常指定中不允许的异常类型, 会发生什么情况?
- 13.19 `try` 块抛出异常时, 其中已经构造的自动对象发生什么情况?

### 自测练习答案

- 13.1 内存不足以满足 `new` 的请求、数组下标超界、运算溢出、除数为 0、无效函数参数。
- 13.2 (a) 异常处理是用于处理不常发生的情况, 通常在程序准备终止时使用。因此, C++ 编译器的编写人员不像对正常应用程序代码一样对其实现最优化性能。(b) 使用传统控制结构的控制流通常比使用异常更清晰更有效。(c) 另一危险是堆栈解退, 异常发生之前分配的资源可能无法释放。(d) 这些“增加的”异常可能打乱真正的错误类型异常。程序员更难跟踪异常种类。例如, 程序处理大量异常时, 无法确定 `catch(...)` 捕获的是什么异常。
- 13.3 库函数很难满足用户特殊需求的错误处理。
- 13.4 退出程序会使其他程序无法使用其资源, 从而造成资源泄漏。
- 13.5 忽略该 `try` 块的异常处理器 (`catch` 块中), 程序在最后一个 `catch` 块后重新执行。
- 13.6 `try` 块之外抛出的异常会使程序终止。
- 13.7 `catch(...)` 能捕获 `try` 块中抛出的所有错误。其优点是可以捕获所有错误, 缺点是 `catch` 没有参数, 无法引用抛出的所有错误中的信息, 无法知道错误原因。
- 13.8 这时匹配搜索会继续到外面一层 `try` 块。这个过程一直继续, 也许最终还是没有任何匹配的异常处理器。这时调用 `terminate` (默认调用 `abort` 终止程序)。可以用 `set_terminate` 的参数提供另一 `terminate` 函数。
- 13.9 执行 `try` 块后面第一个匹配的异常处理器。
- 13.10 这样可以很好地捕获相关类型的异常。
- 13.11 可以用一个异常类和 `catch` 处理器处理一组异常。发生每个异常时, 可以生成具有不同 `private` 数据的异常对象。`catch` 处理器通过检查 `private` 数据区分异常的类型。
- 13.12 `void *`。
- 13.13 要求标准转换的处理器可能出现在具有准确匹配的处理器之前。
- 13.14 不一定, 但它终止抛出异常的块。
- 13.15 异常由导致异常的 `catch` 处理器所在 `try` 块 (如果有) 的相关 `catch` 处理器 (如果有) 处理。
- 13.16 再抛出异常。
- 13.17 提供从函数可抛出异常类型的异常指定表。
- 13.18 调用函数 `unexpected`。
- 13.19 通过堆栈解退过程, 调用每个对象的析构函数。

### 练习

- 13.20 列出文中所列的程序中发生的各种异常条件, 尽量多列几个异常条件。对每个异常条件, 简单描述程序如何用本章介绍的异常处理方法处理这个异常。典型的异常有: 内存不足以满足 `new` 的请求、数组下标超界、运算溢出、除数为 0、无效函数参数。

13.21 什么情况下程序员在定义处理器捕获对象类型时不提供参数名？

13.22 程序包含下列语句：

```
throw;
```

这种语句通常出现在什么地方？如果出现在其他地方，会发生什么情况？

13.23 下列语句通常出现在什么情况下？

```
catch(...) { throw };
```

13.24 比较异常处理与各种其他错误处理方法。

13.25 列出异常处理比传统错误处理方法的优点。

13.26 说明为什么不宜用异常作为程序控制的替换方法。

13.27 说明处理相关异常的方法。

13.28 到本章为止，我们发现处理构造函数所发现的错误比较麻烦。异常处理提供了处理这种错误更好的办法。考虑String类的构造函数。这个构造函数用new取得自由空间。假设new操作失败，说明不用异常时如何处理这种情况，讨论关键点。说明使用异常时如何处理这种情况。说明异常处理的优点在哪里。

13.29 假设程序抛出异常，开始执行相应的异常处理器。再假设异常处理器本身抛出相同的异常。这样会生成无穷递归吗？编写一个C++程序，测试得出的结论。

13.30 用继承方法生成异常基类和各种派生类。然后显示指定基类的catch处理器能捕获派生类的异常。

13.31 列出返回double或int的条件异常。提供一个int catch处理器和一个double catch处理器。说明不管返回double或int，都只执行double catch处理器。

13.32 编写一个C++程序，产生和处理内存溢出错误。程序通过运算符new循环请求生成动态存储空间。

13.33 编写一个C++程序列出调用块中构造的所有对象的析构函数之后再从块中抛出异常。

13.34 编写一个C++程序，演示只调用发生异常之前构造的成员对象的成员对象析构函数。

13.35 编写一个C++程序，演示catch(...)如何捕获任何异常。

13.36 编写一个C++程序，表明异常处理器的顺序很重要。执行的是第一个匹配的异常处理器。编译和运行程序，显示执行不同异常处理器时的不同效果。

13.37 编写一个C++程序，表明构造函数传递构造失败信息给try块后面的异常处理器。

13.38 编写一个C++程序，用异常类的多重继承层次生成要考虑异常处理器顺序的情况。

13.39 用setjmp/longjmp时，程序可以立即从深层嵌套函数调用将控制转移到错误程序。但由于堆栈解退，不能调用嵌套函数调用期间生成的自动对象的析构函数。编写一个C++程序，演示的确没有调用嵌套函数调用期间生成的自动对象的析构函数。

13.40 编写一个C++程序，演示再抛出异常。

13.41 编写一个C++程序，用set\_unexpected对unexpected设置用户自定义函数，再次用set\_unexpected，然后将unexpected复位为原先的函数。编写一个C++程序，测试set\_terminate和terminate。

13.42 编写一个C++程序，显示本身有try块的函数不必捕获try块中产生的每个错误。有些错误留给外层范围处理。

13.43 编写一个C++程序，从深层嵌套函数调用抛出错误，并让调用链所在try块后面的catch处理器捕获这个异常。

## 第14章 文件处理

### 教学目标

- 能够建立、读写和更新文件
- 熟悉顺序访问文件的处理方式
- 熟悉随机访问文件的处理方式
- 指定高性能无格式的 I/O 操作
- 了解格式化与“原始数据”文件处理的差别
- 用随机访问文件处理建立事务处理程序

### 14.1 简介

存储在变量和数组中的数据是临时的,这些数据在程序运行结束后都会消失。文件用来永久地保存大量的数据。计算机把文件存储在二级存储设备中(特别是磁盘存储设备)。本章要讨论怎样用 C++ 程序建立、更新和处理数据文件(包括顺序存储文件和随机访问文件)。我们要比较格式化与“原始数据”文件处理。第 19 章将介绍从 string 而不是从文件输入和输出数据。

### 14.2 数据的层次

计算机处理的所有数据项最终都是 0 和 1 的组合。采用这种组合方式是因为它非常简单,并且能够经济地制造表示两种稳定状态的电子设备(一种状态代表 1,另一种状态代表 0)。计算机所完成的复杂功能仅仅涉及最基本的对 0 和 1 的操作。

0 和 1 可以认为是计算机中的最小数据项,人们称之为“位”(bit)。bit 是 binary digit(二进制数字)的缩写,一个二进制数字是 0 和 1 的两个值之一。计算机电路完成各种简单的位操作,如确定某个位的值、设置某个位的值和反转某个位的值(0 变为 1,1 变为 0)等等。

程序员如果以底层位的形式处理数据会感到很麻烦,所以更喜欢用十进制数字(即 0、1、2、3、4、5、6、7、8 和 9)、字母(即 A~Z 和 a~z)和专门的符号(即 \$、@、%、&、\*、(、)、-、+、"、:、?、/ 等等)处理数据。数字、字母和专门的符号称为“字符”(character)。能够在特定计算机上用来编写程序和代表数据项的所有字符的集合称为“字符集”(character set)。因为计算机只能处理 1 和 0,所以计算机字符集中的每一个字符都是用称为“字节”(byte)的 0、1 序列表示的。目前最常见的是用 8 位构成一个字节。程序员以字符为单位建立程序和数据项,计算机按位模式操作和处理这些字符。

就像字符是由位构成的,域(field)是由字符构成的。一个域就是一组有意义的字符。例如,一个仅仅包含大写字母和小写字母的域可用来表示某人的名字。

计算机处理的数据项构成了“数据的层次”(data hierarchy)。在这个结构中,数据项从位到字符再到域是越来越大、越来越复杂。

记录（即 C 语言中的结构）是由多个域构成（在 C++ 中称为成员）。例如，在一张工资表中，为某个特定雇员建立的一条记录可能是由如下域组成的：

1. 雇员标识号
2. 名字
3. 地址
4. 每小时工资等级
5. 免税申请号
6. 年度收入
7. 联邦税收额等等

因此，一条记录是一组相关的域。在上面的例子中，每一个域都针对同一个雇员。当然，特定的公司会有许多雇员，所以要为每一个雇员建立一个工资表（记录）。一个文件就是一组相关的记录。某个公司的工资表文件通常包含为每一个雇员建立的记录。较小公司的工资表文件可能只包含 22 条记录，而大型公司的工资表文件可能要包含 100 000 条记录。一个机构建立成百上千个文件，而每一个文件又包含几百万甚至几十亿个字符信息，这是不值得奇怪的。图 14.1 反映了数据的层次。

为了检索文件中指定的记录，每个记录中至少要选出一个域作为“记录关键字”（record key）。记录关键字标识了属于某人或某个实体的记录。例如，在本节的工资表记录中，“雇员标识号”通常选作记录关键字。

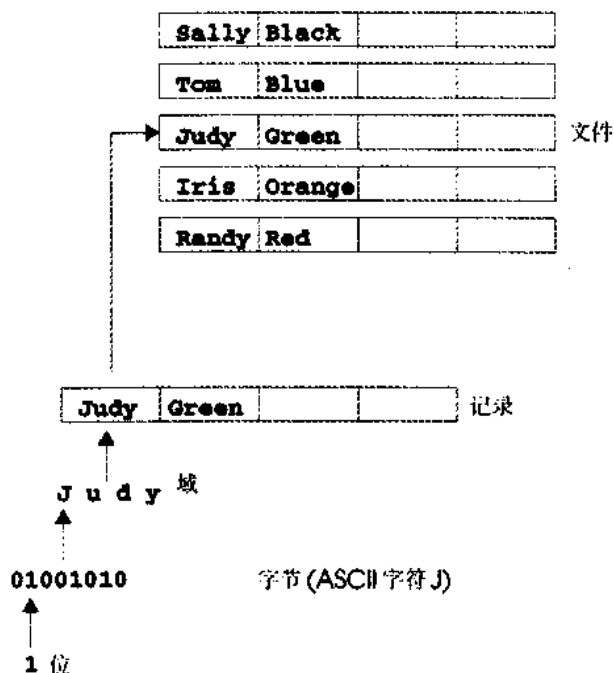


图 14.1 数据的层次

文件中的记录有多种组织方式。最常见的组织方式是按记录关键字字段的顺序存储记录，按这种方式存储记录的文件称为“顺序文件”（sequential file）。在工资表文件中，记录通常按雇员标识号的顺序存储。在第一个雇员的记录中，该雇员的雇员标识号最小，其后的记录中包含的雇员标识号依次递增。

多数商业机构要用许多文件来存储数据。例如，公司里可能要有工资表文件、应收账款目文件（列出客户的欠款）、应付账目文件（列出欠供应商的金额）、存货文件（列出经商的货物）和其他多种类型的文件。有时把一组相关的文件称为“数据库”（database）。为建立和管理数据库而设计的文件集合称为“数据库管理系统”（DBMS）。

### 14.3 文件和流

C++ 语言把每一个文件都看成一个有序的字节流（见图 14.2），每一个文件或者以文件结束符（end-of-file marker）结束，或者在特定的字节号处结束（结束文件的特定的字节号记录在由系统维护和管理的数据结构中）。当打开一个文件时，该文件就和某个流关联起来。第 11 章曾介绍过 `cin`、`cout`、`cerr` 和 `clog` 这 4 个对象会自动生成。与这些对象相关联的流提供程序与特定文件或设备之间的通信通道。例如，`cin` 对象（标准输入流对象）使程序能从键盘输入数据，`cout` 对象（标准输出流对象）使程序能向屏幕输出数据，`cerr` 和 `clog` 对象（标准错误流对象）使程序能向屏幕输出错误消息。

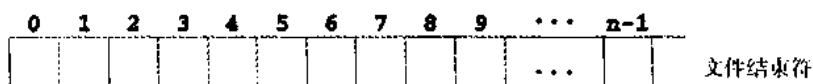


图 14.2 C++ 把文件看成  $n$  个字节

要在 C++ 中进行文件处理，就要包括头文件 `<iostream.h>` 和 `<fstream.h>`。`<fstream.h>` 头文件包括流类 `ifstream`（从文件输入）、`ofstream`（向文件输出）和 `fstream`（从文件输入/输出）的定义。生成这些流类的对象即可打开文件。这些流类分别从 `istream`、`ostream` 和 `iostream` 类派生（即继承它们的功能）。这样，第 11 章“C++ 输入/输出流”中介绍的成员函数、运算符和流操纵算子也可用于文件流。I/O 类的继承关系见图 14.3。

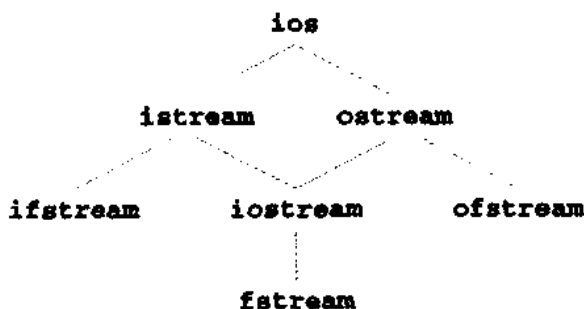


图 14.3 I/O 流类的继承关系

### 14.4 建立顺序访问文件

因为 C++ 把文件看着是无结构的字节流，所以记录等等的说法在 C++ 文件中是不存在的。为此，程序员必须提供满足特定应用程序要求的文件结构。下例说明了程序员是怎样给文件强加一个记录结构。先列出程序，然后再分析细节。

图 14.4 中的程序建立了一个简单的顺序访问文件, 该文件可用在应收账款管理系统中跟踪公司借贷客户的欠款数目。程序能够获取每一个客户的账号、客户名和对客户的结算额。一个客户的数据就构成了该客户的记录。账号在应用程序中用作记录关键字, 文件按账号顺序建立和维护。范例程序假定用户是按账号顺序键入记录的( 为了让用户按任意顺序键入记录, 完善的应收账款管理系统应该具备排序能力)。然后把键入的记录保存并写入文件。

```
1 // Fig. 14.4: fig14_04.cpp
2 // Create a sequential file
3 #include <iostream.h>
4 #include <fstream.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     // ofstream constructor opens file
10    ofstream outClientFile( "clients.dat", ios::out );
11
12    if ( !outClientFile ) { // overloaded ! operator
13        cerr << "File could not be opened" << endl;
14        exit( 1 ); // prototype in stdlib.h
15    }
16
17    cout << "Enter the account, name, and balance.\n"
18         << "Enter end-of-file to end input.\n? ";
19
20    int account;
21    char name[ 30 ];
22    float balance;
23
24    while ( cin >> account >> name >> balance ) {
25        outClientFile << account << ' ' << name
26                     << ' ' << balance << '\n';
27        cout << "? ";
28    }
29
30    return 0; // ofstream destructor closes file
31 }
```

**输出结果:**

```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0
? 400 Stone -42.16
? 500 Rich 224.62
? ^z
```

图 14.4 建立顺序文件

现在我们来研究这个程序。前面曾介绍过, 文件通过建立 `ifstream`、`ofstream` 或 `fstream` 流类的对象而打开。图 14.4 中, 要打开文件以便输出, 因此生成 `ofstream` 对象。向对象构造函数传入两个



参数——文件名和文件打开方式。对于 `ofstream` 对象，文件打开方式可以是 `ios::out`（将数据输出到文件）或 `ios::app`（将数据添加到文件末尾，而不修改文件中现有的数据）。现有文件用 `ios::out` 打开时会截尾，即文件中的所有数据均删除。如果指定文件还不存在，则用该文件名生成这个文件。下列声明（第 10 行）：

```
ofstream outClientFile( "clients.dat", ios::out );
```

生成 `ofstream` 对象 `outClientFile`，与打开输出的文件 `clients.dat` 相关联。参数 `"clients.dat"` 和 `ios::out` 传入 `ofstream` 构造函数，该函数打开文件，从而建立与文件的通信线路。默认情况下，打开 `ofstream` 对象以便输出，因此下列语句：

```
ofstream outClientFile( "clients.dat" );
```

也可以打开 `clients.dat` 进行输出。图 14.5 列出了文件打开方式。

#### 常见编程错误 14.1

打开一个用户想保留数据的现有文件进行输出(`ios::out`方式)。这种操作会删除文件的内容而不会给予警告。

#### 常见编程错误 14.2

用错误的 `ofstream` 对象指明一个文件。

也可以生成 `ofstream` 对象而不打开特定文件，可以在后面再将文件与对象相连接。例如，下列声明：

```
ofstream outClientFile;
```

生成 `ofstream` 对象 `outClientFile`。`ofstream` 成员函数 `open` 打开文件并将其与现有 `ofstream` 对象相连接，如下所示：

```
outClientFile.open( "clients.dat", ios::out );
```

文件打开方式	说明
<code>ios::app</code>	将所有输出写入文件末尾
<code>ios::ate</code>	打开文件以便输出，并移到文件末尾（通常用于添加数据） 数据可以写入文件中的任何地方
<code>ios::in</code>	打开文件以便输入
<code>ios::out</code>	打开文件以便输出
<code>ios::trunc</code>	删除文件现有内容（是 <code>ios::out</code> 的默认操作）
<code>ios::nocreate</code>	如果文件不存在，则文件打开失败
<code>ios::noreplace</code>	如果文件存在，则文件打开失败

图 14.5 文件打开方式

#### 常见编程错误 14.3

在引用文件之前忘记打开该文件。

生成 `ofstream` 对象并准备打开时，程序测试打开操作是否成功。下列 `if` 结构中的操作（第 12 行到第 15 行）：

```
if ( !outClientFile ) {
    cerr << "File could not be opened" << endl;
```

```
    exit( 1 );
}
```

用重载的 `ios` 运算符成员函数 `operator!` 确定打开操作是否成功。如果 `open` 操作的流将 `failbit` 或 `badbit` 设置, 则这个条件返回非 0 值 ( `true` )。可能的错误是试图打开读取不存在的文件、试图打开读取没有权限的文件或试图打开文件以便写入而磁盘空间不足。

如果条件表示打开操作不成功, 则输出错误消息 “File could not be opened”, 并调用函数 `exit` 结束程序, `exit` 的参数返回到调用该程序的环境中, 参数 0 表示程序正常终止, 任何其他值表示程序因某个错误而终止。 `exit` 返回的值让调用环境 (通常是操作系统) 对错误做出相应的响应。

另一个重载的 `ios` 运算符成员函数 `operator void*` 将流变成指针, 使其测试为 0 (空指针) 或非 0 (任何其他指针值)。如果 `failbit` 或 `badbit` (见第 11 章) 对流进行设置, 则返回 0 ( `false` )。下列 `while` 首部的条件自动调用 `operator void*` 成员函数:

```
while ( cin >> account >> name >> balance )
```

只要 `cin` 的 `failbit` 和 `badbit` 都没有设置, 则条件保持 `true`。输入文件结束符设置 `cin` 的 `failbit`。 `operator void*` 函数可以测试输入对象的文件结束符, 而不必对输入对象显式调用 `eof` 成员函数。

如果文件打开成功, 则程序开始处理数据。下列语句 (第 17 行和第 18 行) 提示用户对每个记录输入不同域, 或在数据输入完成时输入文件结束符:

```
cout << "Enter the account, name, and balance.\n"
     << "Enter EOF to and input.\n? ";
```

图 14.6 列出了不同计算机系统中文件结束符的键盘组合。

计算机系统	组合键
UNIX 系统	<ctrl> d
IBM PC 及其兼容机	<ctrl> z
Macintosh	<ctrl> d
VAX (VMS)	<ctrl> z

图 14.6 各种流行的计算机系统中的文件结束组合键

下列语句 (第 24 行):

```
while ( cin >> account >> name >> balance )
```

输入每组数据并确定是否输入了文件结束符。输入文件结束符或不合法数据时, `cin` 的流读取运算符 `>>` 返回 0 (通常这个流读取运算符 `>>` 返回 `cin`), `while` 结构终止。用户输入文件结束符告诉程序没有更多要处理的数据。当用户输入文件结束符组合键时, 设置文件结束符。只要没有输入文件结束符, `while` 结构就一直循环。

第 25 行和第 26 行:

```
outClientFile << account << ' ' << name
              << ' ' << balance << '\n';
```

用流插入运算符 `<<` 和程序开头与文件相关联的 `outClientFile` 对象将一组数据写入文件 “clients.dat”。可以用读取文件的程序取得这些数据 (见 14.5 节)。注意图 14.4 中生成的文件是文本文件, 可以用任何文本编辑器读取。

输入文件结束符后, main 终止, 使得 outClientFile 对象删除, 从而调用其析构函数, 关闭文件 clients.dat。程序员可以用成员函数 close 显式关闭 ofstream 对象, 如下所示:

```
outClientFile.close();
```

#### 性能提示 14.1

程序不再引用的文件应立即显式关闭, 这样可以减少程序继续执行时占用的资源。这种方法还可以提高程序的清晰性。

在图 14.4 的执行范例中, 用户输入了五条记录, 然后键入了表示数据输入结束的文件结束符 (IBM PC 兼容机的屏幕上显示 ^z)。输出结果的对话框中没有说明这些记录究竟是怎样在文件中组织。为了验证文件的建立是成功的, 下一节介绍了读取和打印该文件的程序。

## 14.5 读取顺序访问文件中的数据

为了在需要的时候能够检索要处理的数据, 数据要存储在文件中。上一节演示了怎样建立一个顺序访问的文件。这一节要讨论按顺序读取文件中的数据。

图 14.7 中的程序读取文件 "clients.dat" (图 14.4 中的程序建立) 中的记录, 并打印出了记录的内容。通过建立 ifstream 类对象打开文件以便输入。向对象传入的两个参数是文件名和文件打开方式。下列声明:

```
ifstream inClientFile( "clients.dat", ios::in );
```

生成 ifstream 对象 inClientFile, 并将其与打开以便输入的文件 clients.dat 相关联。括号中的参数传入 ifstream 构造函数, 打开文件并建立与文件的通信线路。

打开 ifstream 类对象默认为进行输入, 因此下列语句:

```
ifstream inClientFile( "clients.dat" );
```

可以打开 clients.dat 以便输入。和 ofstream 对象一样, ifstream 对象也可以生成而不打开特定文件, 然后再将对象与文件相连接。

#### 编程技巧 14.1

如果文件内容不能修改, 那么只能打开文件以便输入 (用 ios::in), 避免不小心改动文件。这是最低权限原则的又一个例子。

程序用 !inClientFile 条件确定文件是否打开成功, 然后再从文件中读取数据。下列语句:

```
while (inClientFile >> account >> name >> balance )
```

从文件中读取一组值 (即记录)。第一次执行完该条语句后, account 的值为 100, name 的值为 "Jones", balance 的值为 24.98。每次执行程序中的该条语句时, 函数都读取文件中的另一条记录, 并把新的值赋给 account、name 和 balance。记录用函数 outputLine 显示, 该函数用参数化流操纵算子将数据格式化之后再显示。到达文件末尾时, while 结构中的输入序列返回 0 (通常返回 inClientFile 流), ifstream 析构函数将文件关闭, 程序终止。

```
1 // Fig. 14.7: fig14_07.cpp
2 // Reading and printing a sequential file
```

```

3 #include <iostream.h>
4 #include <fstream.h>
5 #include <iomanip.h>
6 #include <stdlib.h>
7
8 void outputLine( int, const char *, double );
9
10 int main()
11 {
12     // ifstream constructor opens the file
13     ifstream inClientFile( "clients.dat", ios::in );
14
15     if ( !inClientFile ) {
16         cerr << "File could not be opened\n";
17         exit( 1 );
18     }
19
20     int account;
21     char name[ 30 ];
22     double balance;
23
24     cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
25          << setw( 13 ) << "Name" << "Balance\n";
26
27     while ( inClientFile >> account >> name >> balance )
28         outputLine( account, name, balance );
29
30     return 0; // ifstream destructor closes the file
31 }
32
33 void outputLine( int acct, const char *name, double bal )
34 {
35     cout << setiosflags( ios::left ) << setw( 10 ) << acct
36          << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
37          << resetiosflags( ios::left )
38          << setiosflags( ios::fixed | ios::showpoint )
39          << bal << '\n';
40 }

```

**输出结果：**

Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62

图 14.7 读取并打印一个顺序文件

为了按顺序检索文件中的数据,程序通常要从文件的起始位置开始读取数据,然后连续地读取所有的数据,直到找到所需要的数据为止。程序执行中可能需要按顺序从文件开始位置处理文件中的数据好几次。istream类和ostream类都提供成员函数,使程序把“文件位置指针”(file position pointer,指示读写操作所在的下一个字节号)重新定位。这些成员函数是istream类的seekg(“seek

get”)和 ostream 类的 seekp (“seek put”)。每个 istream 对象有个 get 指针,表示文件中下一个输入相距的字节数,每个 ostream 对象有一个 put 指针,表示文件中下一个输出相距的字节数。下列语句:

```
inClientFile.seekg( 0 );
```

将文件位置指针移到文件开头(位置0),连接 inClientFile。seekg 的参数通常为 long 类型的整数。第二个参数可以指定寻找方向,ios::beg(默认)相对于流的开头定位,ios::cur 相对于流当前位置定位,ios::end 相对于流结尾定位。文件位置指针是个整数值,指定文件中离文件开头的相对位置(也称为离文件开头的偏移量)。下面是一些 get 文件位置指针的例子:

```
// position to the nth byte of fileObject
// assumes ios::beg
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position y bytes back from end of fileObject
fileObject.seekg( y, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

ostream 成员函数 seekp 也可以进行类似的操作。成员函数 tellg 和 tellp 分别返回 get 和 put 指针的当前位置。下列语句将 get 文件位置指针值赋给 long 类型的变量 location。

```
location = fileObject.tellg();
```

图 14.8 中的程序可用来显示无借贷客户、公司债主和欠款客户的清单。公司欠债主的金额用负数表示,欠公司的金额用正数表示。程序显示了一个菜单,管理人员键入前三个选项中的一个可获取某方面的借贷信息。选项 1 显示出无借贷客户的清单,选项 2 显示公司债主的清单,选项 3 显示欠款客户的清单,选项 4 终止程序的运行。输入无效值时只是再次提示更新选择。图 14.9 是程序的简单输出。

```
1 // Fig. 14.8: fig14_08.cpp
2 // Credit inquiry program
3 #include <iostream.h>
4 #include <fstream.h>
5 #include <iomanip.h>
6 #include <stdlib.h>
7
8 enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE,
9                  DEBIT_BALANCE, END };
10 int getRequest();
11 bool shouldDisplay( int, double );
12 void outputLine( int, const char *, double );
13
14 int main()
15 {
16     // ifstream constructor opens the file
17     ifstream inClientFile( "clients.dat", ios::in );
18
19     if ( !inClientFile ) {
```

```

20     cerr << "File could not be opened" << endl;
21     exit( 1 );
22 }
23
24 int request;
25 int account;
26 char name[ 30 ];
27 double balance;
28
29 cout << "Enter request\n"
30     << " 1 - List accounts with zero balances\n"
31     << " 2 - List accounts with credit balances\n"
32     << " 3 - List accounts with debit balances\n"
33     << " 4 - End of run";
34 request = getRequest();
35
36 while ( request != END ) {
37     switch ( request ) {
38         case ZERO_BALANCE:
39             cout << "\nAccounts with zero balances:\n";
40             break;
41         case CREDIT_BALANCE:
42             cout << "\nAccounts with credit balances:\n";
43             break;
44         case DEBIT_BALANCE:
45             cout << "\nAccounts with debit balances:\n";
46             break;
47     }
48
49     inClientFile >> account >> name >> balance;
50
51     while ( !inClientFile.eof() ) {
52         if ( shouldDisplay( request, balance ) )
53             outputLine( account, name, balance );
54
55         inClientFile >> account >> name >> balance;
56     }
57
58     inClientFile.clear(); // reset eof for next input
59     inClientFile.seekg( 0 ); // move to beginning of file
60     request = getRequest();
61 }
62
63 cout << "End of run." << endl;
64
65 return 0; // ifstream destructor closes the file
66 }
67
68 int getRequest()
69 {
70     int request;
71     do {
72         cout << "\n? ";
73         cin >> request;

```

```

76     } while( request < ZERO_BALANCE && request > END );
77
78     return request;
79 }
80
81 bool shouldDisplay( int type, double balance )
82 {
83     if ( type == CREDIT_BALANCE && balance < 0 )
84         return true;
85
86     if ( type == DEBIT_BALANCE && balance > 0 )
87         return true;
88
89     if ( type == ZERO_BALANCE && balance == 0 )
90         return true;
91
92     return false;
93 }
94
95 void outputLine( int acct, const char *name, double bal )
96 {
97     cout << setiosflags( ios::left ) << setw( 10 ) << acct
98         << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
99         << resetiosflags( ios::left )
100        << setiosflags( ios::fixed | ios::showpoint )
101        << bal << '\n';
102 }

```

图 14.8 借贷查询程序

```

Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run.
? 1

Accounts with zero balances:
300      White      0.00

? 2

Accounts with credit balances:
400      Stone     -42.16

? 3

Accounts with debit balances:
100      Jones      24.98
200      Doe        345.67
500      Rich       224.62
? 4
End of run.

```

图 14.9 图 14.8 中程序的示例输出

## 14.6 更新顺序访问文件

14.4 节介绍了格式化和写入顺序访问文件的数据修改时会有破坏文件中其他数据的危险。例如,如果要把名字 "White" 改为 "Worthington", 则不是简单地重定义旧的名字。White 的记录是以如下形式写入文件中的:

```
300 White 0.00
```

如果用新的名字从文件中相同的起始位置重写该记录,记录的格式就成为:

```
300 Worthington 0.00
```

因为新的记录长度大于原始记录的长度,所以从 "Worthington" 的第二个 "o" 之后的字符将重定义文件中的下一条顺序记录。出现该问题的原因在于:在使用流插入运算符 << 和流读取运算符 >> 的格式化输入/输出模型中,域的大小是不定的,因而记录的大小也是不定的。例如,7、14、-117、2047 和 27383 都是 int 类型的值,虽然它们的内部存储占用相同的字节数,但是将它们以格式化文本打印到屏幕上或存储在磁盘上时要占用不同大小的域。因此,格式化输入/输出模型通常不用来更新已有的记录。

也可以修改上述名字,但比较危险。比如,在 300 White 0.00 之前的记录要复制到一个新的文件中,然后写入新的记录并把 300 White 0.00 之后的记录复制到新文件中。这种方法要求在更新一条记录时处理文件中的每一条记录。如果文件中一次要更新许多记录,则可以用这种方法。

## 14.7 随机访问文件

前面介绍了生成顺序访问文件和从顺序访问文件搜索特定信息。顺序访问文件不适合快速访问应用程序,即要立即找到特定记录的信息。快速访问应用程序的例子有航空订票系统、银行系统、销售网点系统、自动柜员机和其他要求快速处理特定数据的事务处理系统 (transaction processing system)。银行要面对成千上万的客户,但自动柜员机能在瞬间作出响应。这种快速访问应用程序是用随机访问文件 (random access file) 实现的。随机访问文件的各个记录可以直接快速访问,而不需要进行搜索。

前面曾介绍过,C++ 不提供文件结构。因此应用程序要自己生成随机访问文件。虽然实现随机访问文件还有其他方法,但是本书的讨论只限于使用定长记录的这种简洁明了的方法。因为随机访问文件中的每一条记录都有相同的长度,所以能够用记录关键字的函数计算出每一条记录相对于文件起始点的位置。不久就会学到怎样立即访问到文件甚至大型文件中指定的记录。

图 14.10 反映了由定长记录组成的随机访问文件的一种观点 (每一条记录为 100 字节)。它就像一列火车,一些车箱是空的,还有一些车箱满载货物,但是火车中的每节车箱具有相同的长度。

可以在不破坏其他数据的情况下把数据插入到随机访问文件中。也能在不重写整个文件的情况下更新和删除以前存储的数据。下面几节要讨论怎样建立随机访问文件、键入数据、顺序和随机地读取数据、更新数据和删除不再需要的数据。



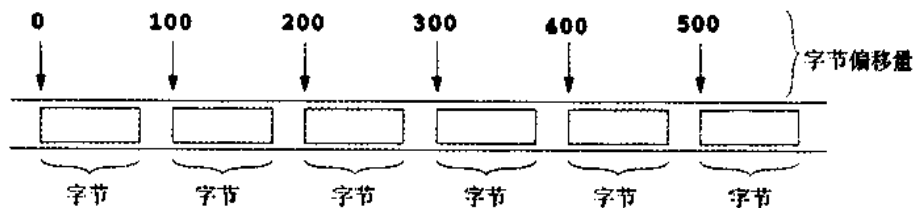


图 14.10 图解具有定长记录的随机访问文件

## 14.8 建立随机访问文件

`ostream` 成员函数 `write` 把从内存中指定位置开始的固定个数的字节送到指定流中, 当流与文件关联时, 数据写入到 “put” 文件位置指针所指示的位置。`istream` 成员函数 `read` 把固定个数的字节从指定流输入到内存中指定地址开始的区域。如果流与文件相关联, 则该字节从 “get” 文件位置指针指定的文件地址开始输入。

现在, 将整型 `number` 写入文件时, 不是用下列语句:

```
outFile << number;
```

对 4 字节整数打印 1 位或 11 位 (10 位加一个符号位, 各要 1 字节存储空间), 而改用:

```
outFile.write( reinterpret_cast<const char*>( &number ),
              sizeof( number ) );
```

这种方法总是写入 4 字节 (在 4 字节整数机器上)。`write` 函数要求一个 `const char*` 类型的参数为第一个参数, 因此我们用 `reinterpret_cast<const char*>` 强制类型转换运算符将 `number` 的地址变为 `const char*` 指针。`write` 的第二个参数是 `size_t` 类型的整数, 指定写入的字节数。可以看出, `istream` 函数 `read` 可以将 4 个字节读回到整型变量 `number` 中。

随机存取文件处理程序很少只把一个域写入文件中, 通常会一次写入一个结构或一个类对象。下面举一个例子。

考虑如下的问题描述:

建立一个能够存储 100 个定长记录的借贷处理系统。每一条记录由账号 (用作记录关键字)、姓、名和借贷金额组成。程序要能够更新、插入和删除一条记录以及能够以格式化文本形式列出所有的记录。要求使用随机访问文件。

以下几节介绍了建立借贷处理程序所需的技术。图 14.11 中的程序说明了怎样打开一个随机访问文件、怎样用 `struct` 定义一条记录格式 (在 `clntdata.h` 头文件中定义) 以及怎样把数据写入磁盘。程序用 `write` 函数和空结构初始化了文件 “credit.dat” 的所有 100 条记录。每一个空结构中, 账号都为 0, 姓氏和名为 `NULL`, 借贷金额为 0.0。文件以这种方式初始化后就在磁盘上建立了存储文件的空间, 并且能够确定某条记录是否包含数据。

在图 14.11 中, 下列语句 (第 34 行到第 36 行):

```
outCredit.write(
    reinterpret_cast<const char*>(&blankClient),
```

```
sizeof( clientData ) );
```

将长度为sizeof(clientData)的blankClient结构写入与ofstream的对象outCredit相关联的文件credit.dat。记住，运算符sizeof返回括号中对象的长度（字节数，见第5章）。注意，第34行函数write的第一个参数应为const char\*类型，但&blankClient的数据类型为clientData\*。要将&blankClient变为相应指针类型，下列表达式：

```
reinterpret_cast<const char *>( &blankClient )
```

用强制类型转换运算符reinterpret\_cast将blankClient地址变为const char\*类型，因此调用write能顺利编译，而不产生语法错误。

```
1 // Fig. 14.11: clntdata.h
2 // Definition of struct clientData used in
3 // Figs. 14.11, 14.12, 14.14 and 14.15.
4 #ifndef CLNTDATA_H
5 #define CLNTDATA_H
6
7 struct clientData {
8     int accountNumber;
9     char lastName[ 15 ];
10    char firstName[ 10 ];
11    float balance;
12 };
13
14 #endif
15 // Fig. 14.11: fig14_11.cpp
16 // Creating a randomly accessed file sequentially
17 #include <iostream.h>
18 #include <fstream.h>
19 #include <stdlib.h>
20 #include "clntdata.h"
21
22 int main()
23 {
24     ofstream outCredit( "credit.dat", ios::out );
25
26     if ( !outCredit ) {
27         cerr << "File could not be opened." << endl;
28         exit( 1 );
29     }
30
31     clientData blankClient = { 0, "", "", 0.0 };
32
33     for ( int i = 0; i < 100; i++ )
34         outCredit.write(
35             reinterpret_cast<const char *>( &blankClient ),
36             sizeof( clientData ) );
37     return 0;
38 }
```

图 14.11 顺序生成随机访问文件

## 14.9 向随机访问文件中随机地写入数据

图 14.12 中的程序把数据写到文件 “credit.dat” 中。ostream 的函数 seekp 和 write 用来将数据存储在文件中指定的位置。程序先用函数 seekp 把 “put” 文件位置指针指向文件中指定的位置，然后用 write 函数写入数据。执行范例见图 14.13。注意图 14.12 的程序包括图 14.11 中定义的头文件 clntdata.h。

```

1 // Fig. 14.12: fig14_12.cpp
2 // Writing to a random access file
3 #include <iostream.h>
4 #include <fstream.h>
5 #include <stdlib.h>
6 #include "clntdata.h"
7
8 int main()
9 {
10     ofstream outCredit( "credit.dat", ios::ate );
11
12     if ( !outCredit ) {
13         cerr << "File could not be opened." << endl;
14         exit( 1 );
15     }
16
17     cout << "Enter account number "
18          << "(1 to 100, 0 to end input)\n? ";
19
20     clientData client;
21     cin >> client.accountNumber;
22
23     while ( client.accountNumber > 0 &&
24            client.accountNumber <= 100 ) {
25         cout << "Enter lastname, firstname, balance\n? ";
26         cin >> client.lastName >> client.firstName
27             >> client.balance;
28
29         outCredit.seekp( ( client.accountNumber - 1 ) *
30                          sizeof( clientData ) );
31         outCredit.write(
32             reinterpret_cast<const char *>( &client ),
33             sizeof( clientData ) );
34
35         cout << "Enter account number\n? ";
36         cin >> client.accountNumber;
37     }
38
39     return 0;
40 }

```

图 14.12 把数据随机地写入随机访问文件中

```

Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance

```

```

? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Barker Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0

```

图 14.13 图 14.12 中程序的示例输出

第 29 行和第 30 行:

```

outCredit.seekp( ( client.accountNumber - 1 ) *
                 sizeof( clientData ) );

```

将 outCredit 对象的 “put” 文件位置指针放在  $(\text{client.accountNumber} - 1) * \text{sizeof}(\text{clientData})$  求出的字节位置处。由于账号在 1 到 100 之间，因此计算记录的字节位置时要从账号减 1。这样，对于记录 1，文件位置指针设置为文件的字节 0。注意 ofstream 的对象 outCredit 用文件打开方式 ios::ate 打开。“put” 文件位置指针最初设置为文件末尾，但数据可以在文件中的任何地方写入。

## 14.10 从随机访问文件中顺序地读取数据

上面几节生成了随机访问文件并将数据写入这个文件中。本节要开发一个程序，顺序读取这个文件，只打印包含数据的记录。该程序还有另一好处，将在本节最后说明，读者不妨先猜猜看。

istream 的函数 read 从指定流的当前位置向对象输入指定字节数。例如，图 14.14 中下列语句：

```

inCredit.read( reinterpret_cast<char*>( &client ),
              sizeof( clientData ) );

```

从与 ifstream 的对象 inCredit 相关联的文件中读取  $\text{sizeof}(\text{clientData})$  指定的字节数，并将数据存放在结构 client 中。注意函数 read 要求第一个参数类型为 char\*。由于 &client 的类型为 clientData\*，因此 &client 要用强制类型转换运算符 reinterpret\_cast 变为 char\*。注意图 14.14 中的程序中包括图 14.11 定义的头文件 clntdata.h。

```

1 // Fig. 14.14: fig14_14.cpp
2 // Reading a random access file sequentially
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <fstream.h>

```

```

6 #include <stdlib.h>
7 #include "clntdata.h"
8
9 void outputLine( ostream&, const clientData & );
10
11 int main()
12 {
13     ifstream inCredit( "credit.dat", ios::in );
14
15     if ( !inCredit ) {
16         cerr << "File could not be opened." << endl;
17         exit( 1 );
18     }
19
20     cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
21          << setw( 16 ) << "Last Name" << setw( 11 )
22          << "First Name" << resetiosflags( ios::left )
23          << setw( 10 ) << "Balance" << endl;
24
25     clientData client;
26
27     inCredit.read( reinterpret_cast<char*>( &client ),
28                  sizeof( clientData ) );
29
30     while ( inCredit && !inCredit.eof() ) {
31
32         if ( client.accountNumber != 0 )
33             outputLine( cout, client );
34
35         inCredit.read( reinterpret_cast<char*>( &client ),
36                      sizeof( clientData ) );
37     }
38
39     return 0;
40 }
41
42 void outputLine( ostream &output, const clientData &c )
43 {
44     output << setiosflags( ios::left ) << setw( 10 )
45           << c.accountNumber << setw( 16 ) << c.lastName
46           << setw( 11 ) << c.firstName << setw( 10 )
47           << setprecision( 2 ) << resetiosflags( ios::left )
48           << setiosflags( ios::fixed | ios::showpoint )
49           << c.balance << '\n';
50 }

```

**输出结果:**

Account	Last Name	First Name	Balance
29	Brown	Nancy	-24.54
33	Dunn	Stacey	314.33
37	Barker	Doug	0.00
88	Smith	Dave	258.34
96	Stone	Sam	34.98

图 14.14 从随机访问文件中顺序地读取数据

图 14.14 的程序顺序读取 "credit.dat" 文件中的每个记录, 检查每个记录中是否包含数据, 并打印包含数据的记录。第 30 行的下列条件:

```
while ( inCredit && !inCredit.eof() ) {
```

用 ios 成员函数 eof 确定是否到达文件末尾, 如果到达文件末尾, 则终止执行 while 结构。如果读取文件时发生错误, 则循环终止, 因为 inCredit 值为 false。从文件中输入的数据用 outputLine 输出, outputLine 取两个参数, 即 ostream 对象和要输出的 clientData 结构。ostream 参数类型可以支持任何 ostream 对象 (如 cout) 或 ostream 的任何派生类对象 (如 ofstream 类型的对象) 作为参数。这样, 同一函数既可输出到标准输出流, 也可输出到文件流, 而不必编写另一个函数。

这些程序还有另一好处, 输出窗口中的记录已经按账号排序列出, 这是用直接访问方法将这些记录存放到文件中的结果。试比较第 4 章介绍的冒泡排序, 用直接访问方法排序显然快多了。这个速度是通过生成足够大的文件来保证完成每个记录而实现的。当然, 大多数时候文件存储都是稀疏的, 因此会浪费存储空间。这是另一个以空间换取时间的例子: 加大空间可以得到更快的排序算法。

## 14.11 实例研究: 事务处理程序

下面介绍一个有实际意义的使用随机访问文件的事务处理程序。该程序维护银行的账目信息。程序能够更新、添加和删除账号, 并且能够把所有当前账号的格式化清单存储在一个用于打印的文本文件中。我们假定已经通过执行图 14.11 中的程序建立了文件 credit.dat, 并用图 14.12 的程序插入了初始值。

程序有五个选项 (第 5 个选项终止程序)。选项 1 调用函数 textFile 把所有的格式化的账号存储在文本文件 print.txt 中 (以后可能要打印这个文件)。函数 textFile 取一个 fstream 对象作为参数, 用于从 credit.dat 文件输入数据。函数 textFile 用 istream 成员函数 read 和图 14.14 介绍的顺序文件访问方法从 credit.dat 输入数据。使用 14.10 节讨论的函数 outputLine 将数据输出到 print.txt 文件。注意 textFile 用 istream 成员函数 seekg 保证文件位置指针在文件开头。选择了选项 1 后, 文件 accounts.txt 中包含如下内容:

Account	Last Name	First Name	Balance
29	Brown	Nancy	-24.54
33	Dunn	Stacey	314.33
37	Barker	Doug	0.00
88	Smith	Dave	258.34
96	Stone	Sam	34.98

选项 2 调用函数 updateRecord 更新账号。该函数只更新已存在的记录, 所以函数首先检查用户指定的记录是否为空。用 istream 成员函数 read 把记录读到结构 client 中, 然后把成员 client.accountNumber 与 0 比较。如果 client.accountNumber 为 0, 说明该条记录中不包含信息, 因此打印出说明该记录为空的消息, 然后再显示出选项菜单。如果记录中包含信息, 函数 updateRecord 用函数 outputLine 在屏上显示记录, 并输入事务金额、计算新的结算结果以及把记录重写到文件中。选项 2 的典型输出如下所示:

```
Enter account to update (1 - 100): 37
37      Barker      Doug      0.00
Enter charge (+) or payment (-): +87.99
```

37	Barker	Doug	87.99
----	--------	------	-------

选项3调用函数 `newRecord` 把新的账号添加到文件中。如果用户键入了一个已有的账号，函数 `newRecord` 显示出说明该账号已存在的消息，并再次显示出选项菜单。函数添加新记录的过程与图 14.12 中的程序所用的方法相同。选项3的典型输出如下所示：

```
Enter new account number(1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

选项4调用函数 `deleteRecord` 删除文件中的一条记录。提示用户输入账号，只能删除已存在的记录，如果该账号的记录为空，函数显示出账号不存在的错误消息。如果存在该账号，通过将空记录 (`blankClient`) 复制到文件中重新初始化该记录。删除记录时会显示一个消息。选项4的典型输出如下所示：

```
Enter account to delete(1 - 100): 29
Account #29 deleted.
```

打开 "credit.dat" 文件时，要用 `ios::in` 和 `ios::out` 的或操作生成 `fstream` 对象以便读写。

```
1 // Fig. 14.15: fig14_15.cpp
2 // This program reads a random access file sequentially,
3 // updates data already written to the file, creates new
4 // data to be placed in the file, and deletes data
5 // already in the file.
6 #include <iostream.h>
7 #include <fstream.h>
8 #include <iomanip.h>
9 #include <stdlib.h>
10 #include "clntdata.h"
11
12 int enterChoice();
13 void textFile( fstream& );
14 void updateRecord( fstream& );
15 void newRecord( fstream& );
16 void deleteRecord( fstream& );
17 void outputLine( ostream&, const clientData & );
18 int getAccount( const char * );
19
20 enum Choices { TEXTFILE = 1, UPDATE, NEW, DELETE, END };
21
22 int main()
23 {
24     fstream inOutCredit( "credit.dat", ios::in | ios::out );
25
26     if ( !inOutCredit ) {
27         cerr << "File could not be opened." << endl;
28         exit ( 1 );
29     }
30
31     int choice;
32
33     while ( ( choice = enterChoice() ) != END ) {
```

```
34
35     switch ( choice ) {
36         case TEXTFILE:
37             textFile( inOutCredit );
38             break;
39         case UPDATE:
40             updateRecord( inOutCredit );
41             break;
42         case NEW:
43             newRecord( inOutCredit );
44             break;
45         case DELETE:
46             deleteRecord( inOutCredit );
47             break;
48         default:
49             cerr << "Incorrect choice\n";
50             break;
51     }
52
53     inOutCredit.clear(); // resets end-of-file indicator
54 }
55
56 return 0;
57 }
58
59 // Prompt for and input menu choice
60 int enterChoice()
61 {
62     cout << "\nEnter your choice" << endl
63         << "1 - store a formatted text file of accounts\n"
64         << "   called \"print.txt\" for printing\n"
65         << "2 - update an account\n"
66         << "3 - add a new account\n"
67         << "4 - delete an account\n"
68         << "5 - end program\n? ";
69
70     int menuChoice;
71     cin >> menuChoice;
72     return menuChoice;
73 }
74
75 // Create formatted text file for printing
76 void textFile( fstream &readFromFile )
77 {
78     ofstream outPrintFile( "print.txt", ios::out );
79
80     if ( !outPrintFile ) {
81         cerr << "File could not be opened." << endl;
82         exit( 1 );
83     }
84
85     outPrintFile << setiosflags( ios::left ) << setw( 10 )
86         << "Account" << setw( 16 ) << "Last Name" << setw( 11 )
87         << "First Name" << resetiosflags( ios::left )
88         << setw( 10 ) << "Balance" << endl;
```



```
89   readFromFile.seekg( 0 );
90
91   clientData client;
92   readFromFile.read( reinterpret_cast<char *>( &client ),
93                     sizeof( clientData ) );
94
95   while ( !readFromFile.eof() ) {
96       if ( client.accountNumber != 0 )
97           outputLine( outPrintFile, client );
98
99       readFromFile.read( reinterpret_cast<char *>( &client ),
100                        sizeof( clientData ) );
101   }
102 }
103
104 // Update an account's balance
105 void updateRecord( fstream &updateFile )
106 {
107     int account = getAccount( "Enter account to update" );
108
109     updateFile.seekg( ( account - 1 ) * sizeof( clientData ) );
110
111     clientData client;
112     updateFile.read( reinterpret_cast<char *>( &client ),
113                    sizeof( clientData ) );
114
115     if ( client.accountNumber != 0 ) {
116         outputLine( cout, client );
117         cout << "\nEnter charge (+) or payment (-): ";
118
119         float transaction;    // charge or payment
120         cin >> transaction;    // should validate
121         client.balance += transaction;
122         outputLine( cout, client );
123         updateFile.seekp( ( account-1 ) * sizeof( clientData ) );
124         updateFile.write(
125             reinterpret_cast<const char *>( &client ),
126             sizeof( clientData ) );
127     }
128     else
129         cerr << "Account #" << account
130              << " has no information." << endl;
131 }
132
133 // Create and insert new record
134 void newRecord( fstream &insertInFile )
135 {
136     int account = getAccount( "Enter new account number" );
137
138     insertInFile.seekg( ( account-1 ) * sizeof( clientData ) );
139
140     clientData client;
141     insertInFile.read( reinterpret_cast<char *>( &client ),
142                       sizeof( clientData ) );
143 }
```

```

144     if ( client.accountNumber == 0 ) {
145         cout << "Enter lastname, firstname, balance\n? ";
146         cin >> client.lastName >> client.firstName
147             >> client.balance;
148         client.accountNumber = account;
149         insertInFile.seekp( ( account - 1 ) *
150                             sizeof( clientData ) );
151         insertInFile.write(
152             reinterpret_cast<const char *>( &client ),
153             sizeof( clientData ) );
154     }
155     else
156         cerr << "Account #" << account
157             << " already contains information." << endl;
158 }
159
160 // Delete an existing record
161 void deleteRecord( fstream &deleteFromFile )
162 {
163     int account = getAccount( "Enter account to delete" );
164
165     deleteFromFile.seekg( (account-1) * sizeof( clientData ) );
166
167     clientData client;
168     deleteFromFile.read( reinterpret_cast<char *>( &client ),
169                         sizeof( clientData ) );
170
171     if ( client.accountNumber != 0 ) {
172         clientData blankClient = { 0, "", "", 0.0 };
173
174         deleteFromFile.seekp( ( account - 1 ) *
175                               sizeof( clientData ) );
176         deleteFromFile.write(
177             reinterpret_cast<const char *>( &blankClient ),
178             sizeof( clientData ) );
179         cout << "Account #" << account << " deleted." << endl;
180     }
181     else
182         cerr << "Account #" << account << " is empty." << endl;
183 }
184
185 // Output a line of client information
186 void outputLine( ostream &output, const clientData &c )
187 {
188     output << setiosflags( ios::left ) << setw( 10 )
189         << c.accountNumber << setw( 16 ) << c.lastName
190         << setw( 11 ) << c.firstName << setw( 10 )
191         << setprecision( 2 ) << resetiosflags( ios::left )
192         << setiosflags( ios::fixed | ios::showpoint )
193         << c.balance << '\n';
194 }
195
196 // Get an account number from the keyboard
197 int getAccount( const char *prompt )
198 {
199     int account;

```

```
200
201     do {
202         cout << prompt << " (1 - 100): ";
203         cin >> account;
204     } while ( account < 1 || account > 100 );
205
206     return account;
207 }
```

图 14.15 银行账目程序

## 14.12 对象的输入/输出

本章和第11章介绍C++的面向对象式的输入/输出。但我们的例子主要考虑传统数据类型的I/O而不是用户自定义类对象的I/O。第8章介绍了如何用运算符重载输入与输出类对象。我们通过对相应的istream重载流读取运算符>>进行对象输入,通过对相应的ostream重载流插入运算符<<进行对象输出。两种情况下都只输入和输出对象的数据成员,而且都是对特定的抽象数据类型对象有意义的方式进行。对象成员函数在计算机内部提供,在数据输入时通过重载流插入运算符而与数据值组合。

对象的数据成员输出到磁盘文件时,就会丢失对象的类型信息。我们存盘的只有数据,而没有类型信息。如果读取这个数据的程序知道其对应的对象类型,则数据读取到该类型的对象。

如果同一文件中存放不同类型的对象,则会发生有趣的问题,如何在读取到程序中时区分它们(或其数据成员集合)呢?当然,问题在于对象通常没有类型域(见第10章“虚函数和多态性”中的介绍)。

一个方法是让每个重载的输出运算符输出类型代码,放在表示一个对象的数据成员集合前面。然后对象输入总是以读取类型代码域开头,并用switch语句调用相应的重载函数。尽管这个方法没有多态编程那么巧妙,但提供了在文件中保持对象并在需要时读取的机制。

### 小结

- 计算机处理的所有数据项最终都是0和1的组合。
- 可以认为计算机中的最小数据项是0和1,该数据项称为“位”。
- 数字、字母和专门的符号称为“字符”。能够在特定计算机上用来编写程序和代表数据项的所有字符的集合称为“字符集”。因为计算机只能处理1和0,所以计算机字符集中的每一个字符都是用称为“字节”的8位0、1模式表示的。
- 一个域就是一组有意义的字符。
- 记录是一组相关的域。
- 每个记录中通常至少要选出一个域作为“记录关键字”。记录关键字标识了文件中属于某人或某个实体的记录。
- 在文件中组织记录的最常用的方法是把记录组织成顺序访问文件。
- 为建立和管理数据库而设计的程序集合称为“数据库管理系统”(DBMS)。
- C++语言把每一个文件都看成一个有序的字节流。
- 每一个文件根据与机器相关的文件结束符结束。

- 流提供文件与程序之间的通信通道。
- 要在 C++ 中进行文件的 I/O 处理, 就要包括头文件 `<iostream.h>` 和 `<fstream.h>`。`<fstream.h>` 首部包括流类 `ifstream`、`ofstream` 和 `fstream` 的定义。
- 文件通过建立 `ifstream`、`ofstream` 或 `fstream` 流类对象而打开。
- 因为 C++ 把文件看着是无结构的字节流, 所以记录等等的说法在 C++ 语言中是不存在的。为此, 程序员必须提供满足特定应用程序要求的文件结构。
- 通过生成 `ofstream` 对象打开文件以便输出。向对象传入两个参数——文件名和文件打开方式。对于 `ofstream` 对象, 文件打开方式可取 `ios::out` (将数据输出到文件) 或 `ios::app` (将数据添加到文件末尾, 而不修改文件中现有的数据)。现有文件用 `ios::out` 打开时会截尾, 即文件中的所有数据均删除。如果指定文件还不存在, 则用该文件名生成这个文件。
- 用 `ios` 运算符成员函数 `operator!` 确定打开操作是否成功。如果 `open` 操作的流将 `failbit` 或 `badbit` 设置, 则这个条件返回非 0 值 (`true`)。
- 程序可以不处理文件、处理一个文件或处理几个文件。每个文件有惟一的名字, 与相应的文件流对象相关联。所有文件处理函数还引用相应对象的文件。
- `istream` 类和 `ostream` 类都提供成员函数, 使程序把“文件位置指针”重新定位。这些成员函数是 `istream` 类的 `seekg` (“seek get”) 和 `ostream` 类的 `seekp` (“seek put”)。每个 `istream` 对象有一个 `get` 指针, 表示文件中下一个输入相距的字节数; 每个 `ostream` 对象有一个 `put` 指针, 表示文件中下一个输出相距的字节数。
- 成员函数 `tellg` 和 `tellp` 分别返回 “get” 和 “put” 指针的当前位置。
- 实现随机访问文件的简便方法是只用定长记录。这样, 程序就可以迅速计算记录相对于文件开头的具体位置。
- 可以在不破坏其他数据的情况下把数据插入到随机访问文件中。也能在不重写整个文件的情况下更新和删除以前存储的数据。
- `ostream` 成员函数 `write` 把从内存中指定位置开始的固定个数的字节送到指定流中, 当流与文件关联时, 数据写入到 “put” 文件位置指针所指示的位置。
- `istream` 成员函数 `read` 把一定的字节数从指定流输入到内存中指定地址开始的区域。该字节从 “get” 文件位置指针指定的文件地址开始输入。
- `write` 函数要求一个 `const char *` 类型的参数为第一个参数, 因此我们用强制类型转换运算符将其其他类型的地址变为 `const char *` 指针。
- 编译时, 一元运算符 `sizeof` 返回括号中对象的长度 (字节数), `sizeof` 返回无符号整数。
- `istream` 函数 `read` 从指定流的当前位置向对象输入指定字节数, `read` 要求第一个参数类型为 `char *`。
- `ios` 成员函数 `eof` 确定是否到达文件末尾, 如果读取文件时发生错误, 则设置文件结束符。

## 术语

alphabetic field 字母域

binary digit 二进制数

bit 位

byte 字节

cerr( standard error unbuffered ) 无缓冲标准错误流

character field 字符域

character set 字符集

cin( standard input ) 标准输入

clog( standard error buffered ) 缓冲标准错误流

close a file 关闭文件

close member function	close 成员函数	开方式	
cout ( standard output )	标准输出	ios::noreplace file open mode	ios::noreplace 文件打开方式
data hierarchy	数据的层次		
database	数据库	istream class	istream 类
database management system ( DBMS )	数据库管理系统	numeric field	数字域
		orstream class	orstream 类
decimal digit	十进制数	open a file	打开文件
end-of-file	文件尾	open member function	open 成员函数
end-of-file marker	文件结束符	operator! member function	operator!成员函数
ends stream manipulator	ends 流操纵算子	operator void * member function	operator void * 成员函数
field	域		
file	文件	ostream class	ostream 类
file name	文件名	output stream	输出流
file position pointer	文件位置指针	random access file	随机访问文件
fstream class	fstream 类	record	记录
fstream.h header file	fstream 头文件	record key	记录关键字
ifstream class	ifstream 类	seekg istream member function	seekg istream 成员函数
in-core I/O	内核 I/O	seekp ostream member function	seekp ostream 成员函数
in-memory I/O	内存 I/O		
input stream	输入流	sequential access file	顺序访问文件
ios::app file open mode	ios::app 文件打开方式	special symbol	特殊符号
ios::ate file open mode	ios::ate 文件打开方式	stream	流
ios::beg seek starting point	ios::beg 寻找开始点	tellg istream member function	tellg istream 成员函数
ios::cur seek starting point	ios::cur 寻找开始点		
ios::end seek starting point	ios::end 寻找开始点	tellp ostream member function	tellp ostream 成员函数
ios::in file open mode	ios::in 文件打开方式		
ios::out file open mode	ios::out 文件打开方式	truncate an existing file	截尾现有文件
ios::trunc file open mode	ios::trunc 文件打开方式		
ios::nocreate file open mode	ios::nocreate 文件打		

## 自测练习

### 14.1 填空:

- 计算机处理的所有数据项最终都是 \_\_\_\_\_ 和 \_\_\_\_\_ 的组合。
- 计算机所能处理的最小数据项称为 \_\_\_\_\_。
- 一个 \_\_\_\_\_ 是一组相关的记录。
- 数字、字母和专门的符号称为 \_\_\_\_\_。
- 一组相关的文件称为 \_\_\_\_\_。
- fstream、ifstream 和 ofstream 文件流类的成员函数 \_\_\_\_\_ 关闭文件。
- istream 成员函数 \_\_\_\_\_ 从指定流中读取一个字符。
- istream 成员函数 \_\_\_\_\_ 和 \_\_\_\_\_ 从指定流中读取一行数据。

- i) `fstream`, `ifstream` 和 `ofstream` 文件流类成员函数 \_\_\_\_\_ 打开一个文件。
  - j) 以随机访问方式读取文件中的数据通常使用 `istream` 成员函数 \_\_\_\_\_。
  - k) `istream` 和 `ostream` 类成员函数 \_\_\_\_\_、\_\_\_\_\_ 把文件位置指针重定位到输入流与输出流中指定的位置。
- 14.2 判断下列说法是否正确。如果不正确, 请说明原因。
- a) 函数 `read` 不能从标准输入流对象 `cin` 读取数据。
  - b) 程序员必须显式地生成 `cin`、`cout`、`cerr` 和 `clog` 对象。
  - c) 程序必须明确地调用函数 `close` 关闭与 `fstream`、`ifstream` 和 `ofstream` 对象相关的文件。
  - d) 如果文件位置指针没有指向顺序访问文件的起始位置, 要从文件起始位置读取数据必须关闭文件然后再打开它。
  - e) `ostream` 成员函数 `write` 能够把数据写入标准输出流 `cout`。
  - f) 更新顺序访问文件中的数据一般不会重定义其他数据。
  - g) 查找随机访问文件中的指定记录不必从头逐条查找。
  - h) 随机访问文件中的记录必须有统一的长度。
  - i) 函数 `seekp` 和 `seekg` 只能定位相对于文件起始点的位置。
- 14.3 用一条语句分别完成下列要求。假定每一条语句用于同一个程序。
- a) 编写一条语句, 打开以便输入数据的文件 `oldmast.dat`, 用 `ifstream` 对象 `inOldMaster`。
  - b) 编写一条语句, 打开以便输入数据的文件 `trans.dat`, 用 `ifstream` 对象 `Transaction`。
  - c) 编写一条语句, 打开以便输出 (以及建立) 数据的文件 `newmast.dat`, 用 `ofstream` 对象 `outNewMaster`。
  - d) 编写一条语句, 读取文件 `oldmast.dat` 中的一条记录。记录是由整数 `accountNum`、字符串 `name` 和浮点数 `currentBalance` 组成的, 用 `ifstream` 对象 `inOldMaster`。
  - e) 编写一条语句, 读取文件 `trans.dat` 中的一条记录。记录是由整数 `accountNum` 和浮点数 `dollarAmount` 组成的, 用 `ifstream` 对象 `inTransaction`。
  - f) 编写一条语句, 向文件 `newmast.dat` 中写入一条记录, 记录是由整数 `accountNum`、字符串 `name` 和浮点数 `currentBalance` 组成的, 用 `ofstream` 对象 `outNewMaster`。
- 14.4 指出下列程序段中的错误, 并说明如何纠正。
- a) 没有打开 `ofstream` 对象 `outPayable` 所引用的文件 (`payables.dat`)。
 

```
outPayable << account << company << amount << endl;
```
  - b) 下面一条语句要从文件 `payables.dat` 中读取一条记录。 `ifstream` 对象 `inPayable` 引用该文件, `istream` 对象 `inReceivable` 引用文件 `receivables.dat`。
 

```
inReceivable >> account >> company >> amount;
```
  - c) 打开文件 `tools.dat`, 在不删除当前数据的情况下把数据添加到文件中。
 

```
ofstream outTools( "tools.dat", ios::out );
```

## 自测练习答案

- 14.1 a) 1、0。 b) 位。 c) 文件。 d) 字符。 e) 数据库。 f) `close`。 g) `get`。 h) `get`、`getline`。 i) `open`。 j) `read`。 k) `seekg`、`seekp`。

- 14.2 a) 不正确。函数 read 可以从 istream 派生的任何输入流对象读取。  
b) 不正确。这 4 个流自动生成。应在文件中包括 <iostream.h> 头文件, 该文件首部包含了对这 4 个流对象的声明。  
c) 不正确。流对象离开范围或程序执行终止前执行 ifstream、ofstream 和 fstream 对象的析构函数时关闭文件, 但作为一个编程技巧, 应在文件不再需要时立即用 close 关闭。  
d) 不正确。成员函数 seekp 或 seekg 可以将 put 或 get 文件位置指针移到文件开头。  
e) 正确。  
f) 不正确。大多数情况下, 顺序文件的记录没有统一的长度。因此, 更新一条记录可能会重定义其他数据。  
g) 正确。  
h) 不正确。随机访问文件中的记录通常具有统一的长度。  
i) 不正确。根据文件位置指针从文件起始点、结束点和当前点定位文件中的位置都是可能的。
- 14.3 a) ifstream inOldMaster( "oldmast.dat", ios::in);  
b) ifstream inTransaction( "trans.dat", ios::in );  
c) ofstream outNewMaster( "newmast.dat", ios::out );  
d) inOldMaster >> accountNum >> name >> currentBalance;  
e) inTransaction >> accountNum >> dollarAmount;  
f) outNewMaster << accountNum << name << currentBalance;
- 14.4 a) 不正确: 文件 "payables.dat" 没打开就试图向流输出数据。  
纠正: 用 ostream 函数 open 打开 "payables.dat" 以便输出。  
b) 不正确: 用不正确的 istream 对象从文件 "payables.dat" 读取记录。  
纠正: 用 istream 对象 inPayable 引用 "payables.dat"。  
c) 不正确: 删除了文件内容, 因为文件打开便会输出(ios::out)。  
纠正: 可以使用打开文件以便更新(ios::ate)或打开文件以便添加(ios::app)的方法将数据添加到文件中。

## 练习

### 14.5 填空:

- a) 计算机把大量的数据存储在二级存储设备(如 \_\_\_\_\_)上。  
b) 一条 \_\_\_\_\_ 是由几个域组成的。  
c) 可以包含数字、字母和空格的域称为 \_\_\_\_\_ 域。  
d) 为了便于检索文件中的某条指定的记录, 每条记录都有一个域被选作 \_\_\_\_\_。  
e) 计算机系统的大多数信息存储在 \_\_\_\_\_ 文件中。  
f) 表达了一定意义的一组相关的字符称为 \_\_\_\_\_。  
g) 头文件 <iostream.h> 声明的标准流对象为 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。  
h) ostream 成员函数 \_\_\_\_\_ 向流中写入一个字符。  
i) ostream 成员函数 \_\_\_\_\_ 通常用来向随机访问文件中写入数据。  
j) istream 成员函数 \_\_\_\_\_ 把文件位置指针重定位。
- 14.6 判断下列说法是否正确。如果不正确, 请说明原因。  
a) 从本质上说, 计算机执行的是对 0 和 1 的操作。

- b) 人们更愿意对位进行操作, 而不愿意操作字符或域, 原因是位更紧凑一些。
- e) 人们把程序和数据项表示成字符, 然后计算机把这些字符作为 0 和 1 的组合进行操作和处理。
- d) 5 位邮政编码是数值域的一个例子。
- e) 在计算机应用程序中, 一个人所住的街道地址通常被看作是由字母组成的域。
- f) 计算机处理的数据项构成了数据的层次。在这个层次结构中, 数据项按域、字符和位的顺序是越来越大, 越来越复杂。
- g) 记录关键字属于某个特定的域, 它能够识别一条记录。
- h) 为了便于计算机处理信息, 多数机构都把他们的信息存储在一个文件中。
- i) C++ 程序总是通过名字来引用文件。
- j) 程序建立一个文件后, 计算机会自动保存这个文件以便将来引用。

14.7 练习 14.3 让读者编写了一组语句, 实际上这些语句正好组成一类重要的文件处理程序(即文件匹配程序)的核心。在商业数据处理中, 通常在每个系统中都有多个文件。例如, 在应收账款系统中, 一般有一个主文件, 它包含了每个顾客的详细信息, 如姓名、地址、电话号码、未付的欠款、信用额度、合同管理, 还可能有近期购买和现金付款的简单记录。

事务发生时(即货物卖出、款额邮到), 这些信息被输入到一个文件中。每个商业周期(有的公司是一个月, 有的是一个星期, 还有的是一天)结束时, 这个事务文件(在练习 14.3 中称为 "trans.dat") 用于主文件(在练习 14.3 中称为 "oldmast.dat") 更新账户的购买和付款记录。每更新一次后, 主文件就被重写成一个新文件("newmast.dat"), 它用于在下一个商业周期快结束时执行更新过程。

文件匹配程序必须处理一些单文件程序中不存在的问题。例如, 并非总是发生匹配, 主文件名某位顾客可能没有在目前的商业周期中购货或付款, 这样在事务文件中就没有这个顾客的记录。类似地, 某位购货或付款的客户可能刚搬到这个社区来, 公司可能来不及为这个顾客建立记录。

以练习 14.3 中的语句为基础, 编写一个完整的文件匹配应收账款目程序。为了进行匹配, 可以用每个文件上的账号作为记录关键字。假设每个文件都是顺序文件, 记录是按账号递增的顺序存储的。

当发生匹配时(即具有相同账号的记录在主文件和事务文件中同时出现), 把事务文件上的美元数加到主文件的当前结算额上, 并把记录写入 newmast.dat 中(假定购货在事务文件中用正数表示, 付款在文件中用负数表示)。当某个特定的账户只有主记录但没有对应的事务记录时, 只把主记录写入 newmast.dat 中。当只有事务记录而没有对应的主记录时, 打印出消息 "Unmatched transaction record for account number..." (在省略号处填入事务记录的账号)。

14.8 编写好练习 14.7 中的程序后, 编写一个简单的程序建立一组检验数据, 测试练习 14.7 中的程序, 使用如下的范例数据:

主文件:		
账号	姓名	结算额
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22



事务文件:	
账号	交易额
100	27.14
300	62.11
400	100.56
900	82.17

- 14.9 用练习 14.8 中建立的检验数据文件运行练习 14.7 中的程序, 用第 14.7 节的程序打印一个新的主文件, 仔细检查结果。
- 14.10 有时 (很常见) 几个事务记录的记录关键字相同, 这是因为一个顾客在同一个商业周期内可能多次购货或付款。重写练习 14.7 中的应收账款文件匹配程序, 使它能够处理具有相同关键字的多个事务记录。修改练习 14.8 中的检验数据, 使它包含以下的事务记录:

账号	美元数
300	83.89
700	80.78
700	1.53

- 14.11 写出完成如下要求的语句。假定已经定义了下面的结构并打开了用于写入数据的随机访问文件。

```
struct person {
    char lastName[15];
    char firstName[15];
    char age[2];
};
```

- 初始化文件 nameage.dat, 使它拥有 100 个 lastName="unsigned"、firstName=""、age="0" 的记录。
  - 输入 10 个姓、名和年龄, 并将它们写入文件。
  - 更新已有信息的文件, 如果没有信息则告诉用户 "No info"。
  - 删除一条已有信息的记录 (可以重新初始化这条记录)。
- 14.12 你是一家五金商店的店主。为了查看工具种类、工具数量以及每件工具的价格, 你需要编制一份商品目录。编写一个程序, 把文件 "hardware.dat" 初始化为 100 条空记录, 输入每件工具的有关数据, 能够列出所有工具的清单、删除某个工具不存在的记录以及更新文件中的任何信息。用工具标识号作为记录号, 在文件中使用下列信息:

记录号	工具名	数量	价格
3	Electric sander	7	57.98
17	Hammer	76	11.99
24	Jig saw	21	11.00
39	Lawn mower	3	79.50
56	Power saw	18	99.99
68	Screwdriver	106	6.99
77	Sledge hammer	11	21.50
83	Wrench	34	7.50

- 14.13 修改第4章编写的电话号码产生程序,使其将输出写入一个文件,这样就可以方便地阅读文件。如果你有计算机化字典,则可以将程序修改成查找字典中的七字母单词。这个程序产生一些有趣的七字母单词组合,该组合可能包括两个或三个单词。例如,电话号码 8432677 产生 "THEBOSS"。将程序修改成用计算机化字典查找字典中的七字母单词,看看是否有一个字母的单词加六个字母的单词、两个字母的单词加五个字母的单词等等。
- 14.14 编写一个程序,用 `sizeof` 运算符确定自己的计算机系统中各种数据类型占用的字节数。为便于以后打印,把结果写入文件 "datasize.dat" 中。文件中的信息格式为:

Data type	Size
char	1
unsigned char	1
short int	2
unsigned short int	2
int	4
unsigned int	4
long int	4
unsigned long int	4
float	4
double	8
long double	16

注意:读者计算机系统中的数据类型大小可能与上面列出的不一样。

## 第15章 数据结构

### 教学目标

- 能够用指针、自引用类和递归构造链式数据结构
- 能够建立和操作链表、队列、堆栈和二叉树等动态数据结构
- 了解链式数据结构的各种重要的应用
- 了解如何用类模拟、继承和复合生成复用数据结构

### 15.1 简介

我们已经研究了定长数据结构，如一维数组、二维数组和结构。本章要介绍执行时大小可变的动态数据结构。链表是连成一行的数据项的集合，可以在链表中的任意位置进行插入和删除操作。堆栈对于编译器和操作系统是非常重要的，插入和删除操作只能在堆栈的一端（顶部）进行。队列是正在“等待”的一行数据，数据的插入发生在队列的尾部、删除发生在队列的头部。二叉树可用于快速查找和排序数据、有效地去除重复的数据项、表示文件系统的目录和把表达式编译成机器语言。这些数据结构还有许多其他的应用。

我们要讨论每一种重要的数据结构，并用范例程序说明怎样建立和操作这些数据结构。我们用类、类模板、继承和复合生成与打包这些数据结构，实现程序的复用性和可维护性。

本章为第20章“标准模板库（STL）”做准备。STL是C++标准库的主要部分，STL提供容器、遍历容器的迭代器和处理容器元素的算法。STL利用本章介绍的各种数据结构并将其打包成模板化的类。STL代码的设计保证了可移植性、有效性和可扩展性。了解本章介绍的数据结构原理与生成之后，就可以更好地利用STL中预打包的数据结构、迭代器和算法。STL是ANSI/ISO草案标准对C++最重要的改进，是世界性的组件集，可以帮助实现复用、复用、再复用。

本章的例子是实用例子，可以在更高级的课程和工业应用中直接采用。这些程序大量使用指针操作。练习中包括丰富的、有用的应用程序。

我们诚恳地希望读者能够学习并完成本章专题“建立自己的编译器”中介绍的内容。我们一直在用编译器把C程序翻译成机器语言，从而能够在计算机上执行我们的程序。在这个专题中，读者要实际建立自己的编译器。该编译器要读取一个语句文件，语句是用简单而且功能强大的高级语言编写的，该语言类似于早期流行的BASIC语言。编译器把这些语句编译成SML指令文件。SML语言在第5章的专题“建立自己的计算机”中已介绍过。然后读者的Simpletron模拟器程序要执行自己的编译器所生成的SML语言程序。这个专题可练习在本课程中所学的大多数知识，它将谨慎地带领读者涉足高级语言的技术规范，并描述了把高级语言的每种类型转换为机器语言指令所需的算法。如果读者勇于面对挑战，可以使用练习中建议的方法尝试增强编译器和Simpletron模拟器的功能。

## 15.2 自引用类

自引用类包含一个指针成员，该指针指向属于同一个类型的类对象。例如，如下类定义：

```
class Node {  
public:  
    Node( int );  
    void setData( int );  
    int getData() const;  
    void setNextPtr( const Node * );  
    const Node *getNextPtr() const;  
private:  
    int data;  
    Node *nextPtr;  
};
```

定义了类型 Node。类型 Node 的结构有两个私有成员，一个是整数成员 data，另一个是指针成员 nextPtr，而指针成员 nextPtr 又指向正在声明为的 Node 类型的结构，所以把这种类称为自引用类。成员 nextPtr 称为“链节”（link），即 nextPtr 可用来把一个 Node 类型的对象与另一个同类型的对象链在一起。Node 类型还有五个成员函数：构造函数接受一个整数用于初始化成员 data，setData 函数设置成员 data 的值，getData 函数返回成员 data 的值，setNextPtr 函数设置成员 nextPtr 的值，getNextPtr 函数返回成员 nextPtr 的值。

可以把自引用结构类链在一起构成有用的数据结构，如链表、队列、堆栈和树。图 15.1 表示了两个链在一起构成一个链表的自引用类对象，图中把表示空（null，即 0）指针的斜杠放在第二个自引用类对象的链节成员位置上，表示该链节不指向另一个结构。空指针通常表示一个数据结构的结尾，就好像空字符（'\0'）表示字符串的结尾一样。

### 常见编程错误 15.1

没有把最后一个节点的链节设置为空（0）。



图 15.1 两个链在一起的自引用类对象

## 15.3 动态内存分配

建立和维护动态数据结构需要实现动态内存分配，即程序在执行时为了链接新的节点能够获得更多的内存空间以及能够释放不再需要的节点。动态内存分配的极限是计算机中可用的物理内存数量，或者是虚拟存储系统中可用的虚拟内存数量。这个数量通常是很小的，因为可用内存必须供多个用户共享。

运算符 new 和 delete 对于实现动态内存分配是很重要的。运算符 new 的参数是被分配的对象类型，它返回指向被分配对象类型的指针。例如，下列语句：

```
Node *newPtr = new Node( 10 );
```

分配 `sizeof(Node)` 个字节数的新的内存区域, 并把指针存储在变量 `newPtr` 中。如果没有可用的内存, `new` 返回 0 指针。10 是节点对象的数据。

函数 `delete` 释放 `new` 分配的内存, 即把所占用的内存交还给系统, 以便以后能够重新分配。如下语句可释放上述 `new` 调用而动态分配的内存:

```
delete newPtr;
```

注意, `newPtr` 本身并不删除, 而是删除 `newPtr` 所指的空间。如果 `newPtr` 值为 0 (表示指向空), 则上述语句无效。

下面几节要讨论链表、堆栈、队列和树。这些数据结构都是用动态内存分配和自引用类建立和维护的。

#### 可移植性提示 15.1

类对象的大小不一定是其成员大小之和。这是因为有些机器要求存储内容沿存储单元的边界开始存储 (见第 16 章)。用 `sizeof` 运算符确定对象大小。

#### 常见编程错误 15.2

认为结构的大小就是其成员大小的和。

#### 常见编程错误 15.3

没有释放不再需要的内存会使系统过早地用完内存。有时把这种现象称为“内存泄漏” (memory leak)。

#### 编程技巧 15.1

使用 `new` 时, 用 `delete` 函数把不再需要的内存立即交还给系统。

#### 常见编程错误 15.4

用 `delete` 释放不是用 `new` 动态分配的内存。

#### 常见编程错误 15.5

引用已经释放的内存。

#### 常见编程错误 15.6

删除已经删除的内存可能在执行时造成无法预料的结果。

## 15.4 链表

链表是用链节指针链在一起的自引用类对象 (称为“节点”) 的线性集合。链表是通过指向链表第一个节点的指针访问的, 其后的节点是通过节点中的链节指针成员访问的。通常, 链表的最后一个节点中的链节指针被设置为空 (表示链尾)。链表的每一个节点是在需要的时候建立的, 链表中的数据是动态存储的。节点中可包含任何类型的数据 (包括其他类对象)。如果节点包含基类指针或继承关系中基类与派生类对象的基类引用, 则可以有这种节点的链表, 并用虚函数调用多态处理这些对象。堆栈和队列也是线性数据结构, 它们是有一些限制的链表。树是非线性数据结构。

虽然可以把一系列数据存储于数组中, 但是用链表存储这些数据有许多优点。在数据元素的个数不可预知时, 使用链表是合适的。链表是动态的, 所以可在需要的时候增加和减小其长度。而数组是在编译时分配内存的, 所以数组的大小是不可改变的。数组的空间很快就能用完, 而链表是在系统没有足够的内存满足动态分配存储空间的要求时才会达到全满的状态。

**性能提示 15.1**

虽然可以把数组元素的个数声明得比预计数据项个数多,但是会浪费内存。链表能够更好地利用内存。

可以在链表的合适的位置插入新的元素,然后按照新的顺序维护链表,已存在的节点不必移动。

**性能提示 15.2**

数组的插入和删除操作是费时的,因为插入点和删除点之后的所有的元素必须都移动合适的位置。

**性能提示 15.3**

数组中的元素在内存中是连续存放的。因为能够根据相对于数组起始位置计算出数组元素的地址,所以能够立即访问到任何数组元素。不能立即访问到链表中的元素。

链表的节点在内存中通常不是连续存放的。但是,链表的节点在逻辑上是连续的。图 15.2 表示有多个节点的链表。

**性能提示 15.4**

为数据结构动态地分配内存能够节省内存,但是指针需要占用存储空间,并且动态分配内存需要一些函数调用的开销。

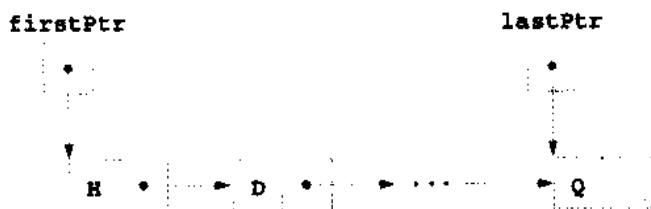


图 15.2 链表的图示

图 15.3 中的程序(输出见图 15.4)用 List 类模板(见第 12 章)操作一系列整数值和一系列浮点数值。驱动程序(fig15\_03.cpp)提供了 5 个选项: 1)在链表开头插入一个值(函数 insertAtFront); 2)在链表末尾插入一个值(函数 insertAtBack); 3)在链表开头删除一个值(函数 removeFromFront); 4)在链表末尾删除一个值(函数 removeFromBack); 5)终止链表处理。下面将详细介绍这个程序,练习 15.20 要求实现一个递归函数,逆向打印链表;练习 15.21 要求实现一个递归函数,查找链表中的特定项目。

图 15.3 包括两个类模板 ListNode 和 List。每个 List 对象中封装了 ListNode 对象的链表。ListNode 类模板包括 private 成员 data 和 nextPtr。ListNode 成员 data 存放 NODETYPE 类型的值,类型参数传入类模板。ListNode 成员 nextPtr 存放链表中下一个 ListNode 对象的指针。

```

1 // Fig. 15.3: listnd.h
2 // ListNode template definition
3 #ifndef LISTND_H
4 #define LISTND_H
5
6 template< class NODETYPE > class List; // forward declaration
7
8 template<class NODETYPE>
9 class ListNode {
10     friend class List< NODETYPE >; // make List a friend
11 public:
12     ListNode( const NODETYPE & ); // constructor

```

```

13     NODETYPE getData() const;        // return data in the node
14 private:
15     NODETYPE data;                    // data
16     ListNode< NODETYPE > *nextPtr;    // next node in the list
17 };
18
19 // Constructor
20 template<class NODETYPE>
21 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
22     : data( info ), nextPtr( 0 ) { }
23
24 // Return a copy of the data in the node
25 template< class NODETYPE >
26 NODETYPE ListNode< NODETYPE >::getData() const { return data; }
27
28 #endif
29 // Fig. 15.3: list.h
30 // Template List class definition
31 #ifndef LIST_H
32 #define LIST_H
33
34 #include <iostream.h>
35 #include <assert.h>
36 #include "listnd.h"
37
38 template< class NODETYPE >
39 class List {
40 public:
41     List();        // constructor
42     ~List();       // destructor
43     void insertAtFront( const NODETYPE & );
44     void insertAtBack( const NODETYPE & );
45     bool removeFromFront( NODETYPE & );
46     bool removeFromBack( NODETYPE & );
47     bool isEmpty() const;
48     void print() const;
49 private:
50     ListNode< NODETYPE > *firstPtr;    // pointer to first node
51     ListNode< NODETYPE > *lastPtr;    // pointer to last node
52
53     // Utility function to allocate a new node
54     ListNode< NODETYPE > *getNewNode( const NODETYPE & );
55 };
56
57 // Default constructor
58 template< class NODETYPE >
59 List< NODETYPE >::List() : firstPtr( 0 ), lastPtr( 0 ) { }
60
61 // Destructor
62 template< class NODETYPE >
63 List< NODETYPE >::~~List()
64 {
65     if ( !isEmpty() ) {        // List is not empty
66         cout << "Destroying nodes ...\n";
67
68         ListNode< NODETYPE > *currentPtr = firstPtr, *tempPtr;

```

```
69
70     while ( currentPtr != 0 ) { // delete remaining nodes
71         tempPtr = currentPtr;
72         cout << tempPtr->data << '\n';
73         currentPtr = currentPtr->nextPtr;
74         delete tempPtr;
75     }
76 }
77
78 cout << "All nodes destroyed\n\n";
79 )
80
81 // Insert a node at the front of the list
82 template< class NODETYPE >
83 void List< NODETYPE >::insertAtFront( const NODETYPE &value )
84 {
85     ListNode< NODETYPE > *newPtr = getNewNode( value );
86
87     if ( isEmpty() ) // List is empty
88         firstPtr = lastPtr = newPtr;
89     else {           // List is not empty
90         newPtr->nextPtr = firstPtr;
91         firstPtr = newPtr;
92     }
93 }
94
95 // Insert a node at the back of the list
96 template< class NODETYPE >
97 void List< NODETYPE >::insertAtBack( const NODETYPE &value )
98 {
99     ListNode< NODETYPE > *newPtr = getNewNode( value );
100
101     if ( isEmpty() ) // List is empty
102         firstPtr = lastPtr = newPtr;
103     else {           // List is not empty
104         lastPtr->nextPtr = newPtr;
105         lastPtr = newPtr;
106     }
107 }
108
109 // Delete a node from the front of the list
110 template< class NODETYPE >
111 bool List< NODETYPE >::removeFromFront( NODETYPE &value )
112 {
113     if ( isEmpty() )           // List is empty
114         return false;          // delete unsuccessful
115     else {
116         ListNode< NODETYPE > *tempPtr = firstPtr;
117
118         if ( firstPtr == lastPtr )
119             firstPtr = lastPtr = 0;
120         else
121             firstPtr = firstPtr->nextPtr;
122
123         value = tempPtr->data; // data being removed
124         delete tempPtr;
```



```
125         return true;           // delete successful
126     }
127 }
128
129 // Delete a node from the back of the list
130 template< class NODETYPE >
131 bool List< NODETYPE >::removeFromBack( NODETYPE &value )
132 {
133     if ( isEmpty() )
134         return false;    // delete unsuccessful
135     else {
136         ListNode< NODETYPE > *tempPtr = lastPtr;
137
138         if ( firstPtr == lastPtr )
139             firstPtr = lastPtr = 0;
140         else {
141             ListNode< NODETYPE > *currentPtr = firstPtr;
142
143             while ( currentPtr->nextPtr != lastPtr )
144                 currentPtr = currentPtr->nextPtr;
145
146             lastPtr = currentPtr;
147             currentPtr->nextPtr = 0;
148         }
149
150         value = tempPtr->data;
151         delete tempPtr;
152         return true;    // delete successful
153     }
154 }
155
156 // Is the List empty?
157 template< class NODETYPE >
158 bool List< NODETYPE >::isEmpty() const
159     { return firstPtr == 0; }
160
161 // Return a pointer to a newly allocated node
162 template< class NODETYPE >
163 ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
164                                     const NODETYPE &value )
165 {
166     ListNode< NODETYPE > *ptr =
167         new ListNode< NODETYPE >( value );
168     assert( ptr != 0 );
169     return ptr;
170 }
171
172 // Display the contents of the List
173 template< class NODETYPE >
174 void List< NODETYPE >::print() const
175 {
176     if ( isEmpty() ) {
177         cout << "The list is empty\n\n";
178         return;
179     }
180 }
```

```
181     ListNode< NODETYPE > *currentPtr = firstPtr;
182
183     cout << "The list is: ";
184
185     while ( currentPtr != 0 ) {
186         cout << currentPtr->data << ' ';
187         currentPtr = currentPtr->nextPtr;
188     }
189
190     cout << "\n\n";
191 }
192
193 #endif
194 // Fig. 15.3: fig15_03.cpp
195 // List class test
196 #include <iostream.h>
197 #include "list.h"
198
199 // Function to test an integer List
200 template< class T >
201 void testList( List< T > &listObject, const char *type )
202 {
203     cout << "Testing a List of " << type << " values\n";
204
205     instructions();
206     int choice;
207     T value;
208
209     do {
210         cout << "? ";
211         cin >> choice;
212
213         switch ( choice ) {
214             case 1:
215                 cout << "Enter " << type << ": ";
216                 cin >> value;
217                 listObject.insertAtFront( value );
218                 listObject.print();
219                 break;
220             case 2:
221                 cout << "Enter " << type << ": ";
222                 cin >> value;
223                 listObject.insertAtBack( value );
224                 listObject.print();
225                 break;
226             case 3:
227                 if ( listObject.removeFromFront( value ) )
228                     cout << value << " removed from list\n";
229
230                 listObject.print();
231                 break;
232             case 4:
233                 if ( listObject.removeFromBack( value ) )
234                     cout << value << " removed from list\n";
235
236                 listObject.print();
```

```

237         break;
238     }
239 } while ( choice != 5 );
240
241     cout << "End list test\n\n";
242 }
243
244 void instructions()
245 {
246     cout << "Enter one of the following:\n"
247         << " 1 to insert at beginning of list\n"
248         << " 2 to insert at end of list\n"
249         << " 3 to delete from beginning of list\n"
250         << " 4 to delete from end of list\n"
251         << " 5 to end list processing\n";
252 }
253
254 int main()
255 {
256     List< int > integerList;
257     testList( integerList, "integer" ); // test integerList
258
259     List< float > floatList;
260     testList( floatList, "float" );    // test integerList
261
262     return 0;
263 }

```

图 15.3 操作链表

List 类模板包括 private 成员 firstPtr (List 对象中第一个 ListNode 的指针) 和 lastPtr (List 对象中最后一个 ListNode 的指针)。默认构造函数将这两个指针初始化为 0 (null)。析构函数保证 List 对象中的所有 ListNode 对象均在删除 List 对象时删除。List 类模板的主要函数为 insertAtFront、insertAtBack、removeFromFront 和 removeFromBack。

isEmpty 函数是个判定函数, 不改变 List 值, 只是确定 List 是否为空 (即 List 的第一个节点指针是否为空)。如果 List 是空表, 则返回 true, 否则返回 false。函数 print 显示 List 的内容。

#### 编程技巧 15.2

把空值 (0) 赋给新节点的链节成员。指针应该在使用前初始化。

```

Testing a List of integer values
Enter one of the following:
 1 to insert at beginning of list
 2 to insert at end of list
 3 to delete from beginning of list
 4 to delete from end of list
 5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

```

```
? 2
Enter integer: 3
The list is: 2 1 3

? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test
Testing a List of float values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter float: 1.1
The list is: 1.1

? 1
Enter float: 2.2
The list is: 2.2 1.1
? 2
Enter float: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter float: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
```

```

The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed

```

图 15.4 图 15.3 程序的示例输出

下面几页要介绍 List 类中每个成员函数的细节。函数 insertAtFront (图 15.5) 将新节点放在链表开头。这个函数完成下列操作步骤:

1. 调用函数 getNewNode 传入 value, 为所插入节点值的常量引用。
2. 函数 getNewNode 用运算符 new 生成新节点, 并返回这个链表节点的指针。如果指针为非 0 值, 则 getNewNode 在 insertAtFront 中返回新分配节点指针给 newPtr。
3. 如果是空表, 则 firstPtr 和 lastPtr 都设置为 newPtr。
4. 如果不是空表, 则通过把 firstPtr 复制到 newPtr->nextPtr 将 newPtr 所指的节点串联到链表中, 因此新节点指向原先链表第一个节点, 并将 newPtr 复制到 firstPtr, 使 firstPtr 指向新的链表第一个节点。

图 15.5 演示了函数 insertAtFront。图中的 a) 显示链表和 insertAtFront 操作之前的新节点。b) 中的虚线箭头表示 insertAtFront 操作的第 2 步和第 3 步, 使得包含 12 的节点成为新的表头。

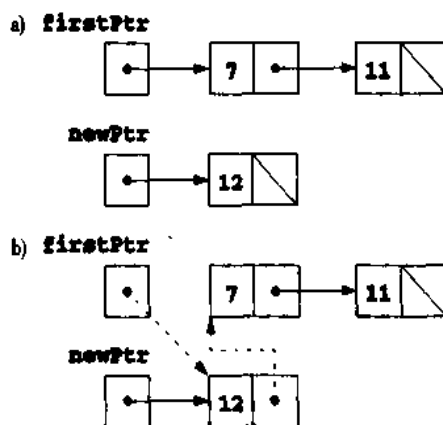


图 15.5 insertAtFront 操作的图形表示

函数 `insertAtBack` (图 15.6) 将新节点放在表末尾, 函数完成下列步骤:

1. 调用函数 `getNode` 传入 `value`, 为所插入节点值的常量引用。
2. 函数 `getNode` 用运算符 `new` 生成新节点, 并返回这个链表节点的指针。如果指针为非 0 值, 则 `getNode` 在 `insertAtFront` 中返回新分配节点指针给 `newPtr`。
3. 如果是空表, 则 `firstPtr` 和 `lastPtr` 都设置为 `newPtr`。
4. 如果不是空表, 则通过把 `newPtr` 复制到 `lastPtr->nextPtr` 将 `newPtr` 所指的节点串联到链表中, 使新节点指向原先链表最后一个节点, 并将 `newPtr` 复制到 `lastPtr`, 使 `lastPtr` 指向新的链表最后一个节点。

图 15.6 演示了函数 `insertAtBack`。图中的 a) 显示链表和 `insertAtFront` 操作之前的新节点。b) 中的虚线箭头表示 `insertAtFront` 操作使得新结点成为非空表表尾的步骤。

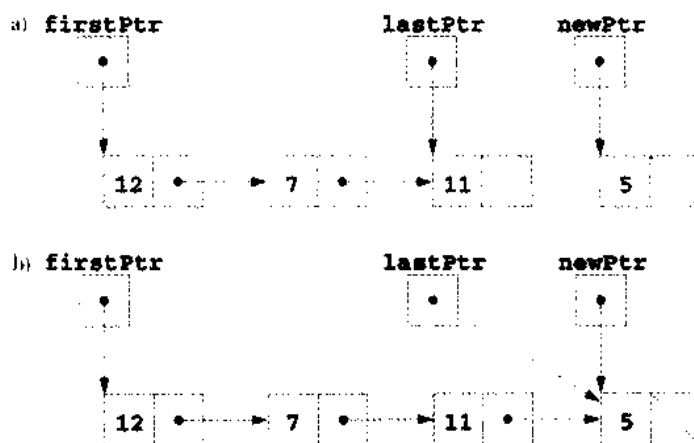


图 15.6 `insertAtBack` 操作的图形表示

函数 `removeFromFront` (图 15.7) 从表中删除开头节点, 并将节点值复制到引用参数中。如果想从空表中删除节点, 则函数返回 `false`, 如果删除成功, 则返回 `true`。函数完成下列步骤:

1. 实例化 `tempPtr` 作为 `firstPtr` 的副本。 `tempPtr` 最终要删除所删节点的内存空间。
2. 如果 `firstPtr` 等于 `lastPtr`, 即表中原先只有一个元素, 则将 `firstPtr` 和 `lastPtr` 设置为 0, 使该节点与链表断开 (使链表变成空表)。
3. 如果表中原先有多个元素, 则保持 `lastPtr` 不变, 只把 `firstPtr` 设置为 `firstPtr->nextPtr`, 即将 `firstPtr` 修改成指向原先的第二个节点 (新的第一个节点)。
4. 完成这些指针操作之后, 向引用参数 `value` 复制所删除节点的 `data` 成员。
5. 现在删除 `tempPtr` 所指节点的内存空间。
6. 返回 `true` 表示删除成功。

图 15.7 演示了函数 `removeFromFront`。图中的 a) 演示删除之前的链表。b) 表示实际指针操作。

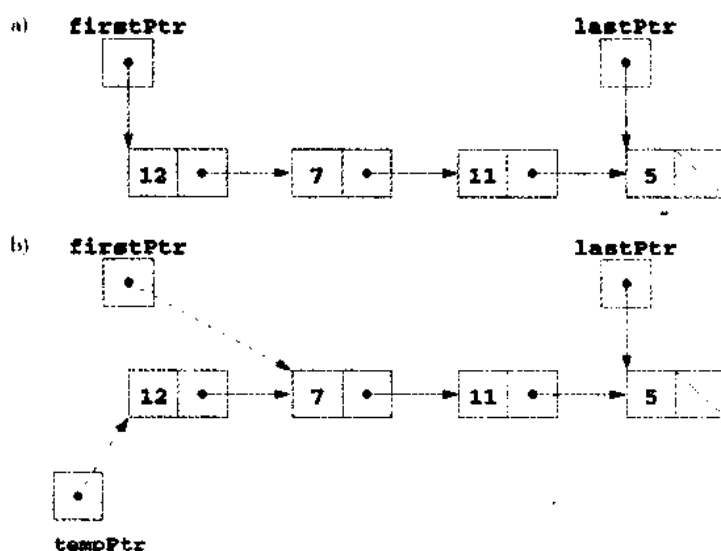


图 15.7 removeFromFront 操作的图形表示

函数 removeFromBack (图 15.8) 从表中删除末尾节点, 并将节点值复制到引用参数中。如果想从空表中删除节点, 则函数返回 false, 如果删除成功, 则返回 true。函数完成下列步骤:

1. 实例化 tempPtr 作为 lastPtr 的副本。tempPtr 最终要删除所删节点的内存空间。
2. 如果 firstPtr 等于 lastPtr, 即表中原先只有一个元素, 则将 firstPtr 和 lastPtr 设置为 0, 使该节点与链表断开 (使链表变成空表)。
3. 如果表中原先有多个元素, 则将 currentPtr 实例化为 firstPtr 的副本。
4. 现在要用 currentPtr 遍历链表, 直到访问倒数第二个节点。这是用 while 循环进行的, 不断在 currentPtr->nextPtr 不等于 lastPtr 时将 currentPtr 换成 currentPtr->nextPtr。
5. 将 currentPtr 复制到 lastPtr, 将最后一个节点与链表断开。
6. 在链表的新末尾结点中, 将 currentPtr->nextPtr 设置为 0。
7. 完成这些指针操作之后, 向引用参数 value 复制所删除节点的数据成员。
8. 现在删除 tempPtr 所指节点的内存空间。
9. 返回 true 表示删除成功。

图 15.8 演示了函数 removeFromBack。图中的 a) 演示删除之前的链表。b) 表示实际指针操作。

函数 print 首先确定链表是否是空的。如果为空, 打印出消息 "The List is empty" 并终止, 否则打印链表中的数据。函数初始化 currentPtr 为 firstPtr 的副本, 并打印 "The list is:". 如果 currentPtr 不为空, 函数就打印出 currentPtr->data, 并把 currentPtr->nextPtr 赋给 currentPtr。注意, 如果链表最后一个节点中的链节不为空, 打印算法就会试图打印链尾之后的内容并发生错误。链表、堆栈和队列的打印算法是相同的。

前面介绍的链表类型是单向链表 (singly-linked list), 链表以第一个节点的指针开始, 每个节点包含下一个顺序节点的指针。这个链表在节点的指针成员为 0 时终止。单向链表只能单向遍历。

循环单向链表 (circular, singly-linked list) 以第一个节点的指针开始, 每个节点包含下一个顺序节点的指针, 最后一个节点不是包含 0 指针, 而是指回第一个节点, 从而形成循环。

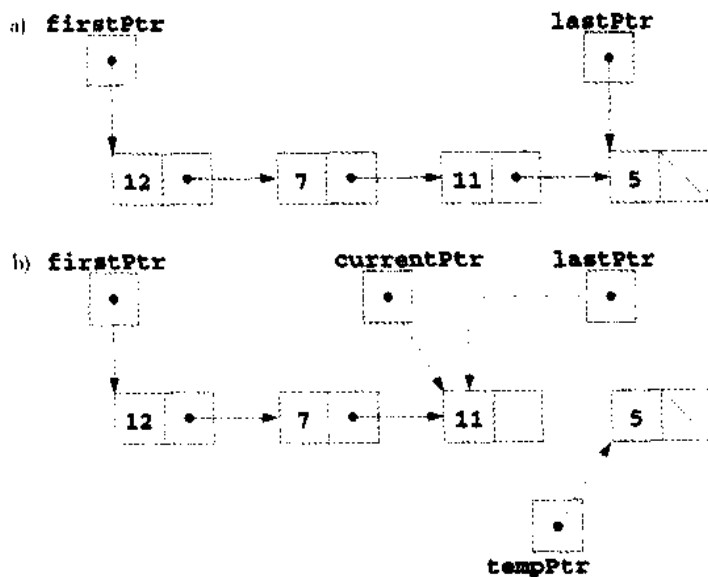


图 15.8 removeFromBack 操作的图形表示

双向链表 (doubly-linked list) 可以正向和逆向遍历。这种链表通常实现为两个开始指针，一个指向链表第一个元素，以便由前到后遍历，一个指向链表最后一个元素，以便由后到前遍历。每个节点都有一个正向指针，指向往前的一个节点，还有一个逆向指针，指向往后的一个节点。例如，如果链表中是按字母排列的电话目录，则查找字母顺序在前的人名时，可以用正向查找，而查找字母顺序在后的人名时，可以用逆向查找。

在循环双向链表 (circular, doubly-linked list) 中，最后一个节点的正向指针指向第一个节点，第一个节点的逆向指针指向最后一个节点，从而形成循环。

## 15.5 堆栈

第 12 章“模板”中介绍了用基础数组实现方法的堆栈类模板。本节介绍用基础指针链表实现方法的堆栈类模板。第 20 章“标准模板库 (STL)”中将介绍堆栈。

堆栈是一种受限制的链表，节点的添加和删除只能在栈顶进行。因此，堆栈被称为“后进先出” (last-in, first-out; LIFO) 的数据结构。堆栈是用指向栈顶元素的指针引用的。堆栈最后一个节点中的链节成员被设置为表示栈底的空 (0)。

### 常见编程错误 15.7

没有把栈底节点中的链节设置为空。

操作堆栈的主要的函数有 push 和 pop。函数 push 建立一个新的节点并把它放在栈顶。函数 pop 删除栈顶节点，将弹出的值存在一个引用变量中，并将该变量传递给调用函数，在 pop 操作成功时返回 true，否则返回 false。

堆栈有许多有趣的应用。例如，不论何时调用一个函数，被调用函数必须知道怎样返回到其调用者，因此要把返回地址压入堆栈。如果发生一系列的函数调用，就以后进先出的方式连续地把返



回值压入堆栈,从而使每个函数都能够返回到其调用者。堆栈支持递归函数的调用,调用方式与通常的非递归调用方式一样。

堆栈包含为函数的自动变量所建立的空间。当函数返回到其调用者时,为函数自动变量所建立的空间从堆栈中弹出,这些变量不再为程序所识别。

编译器在计算表达式和产生机器语言代码的过程中也要用到堆栈。练习中探讨了堆栈的某些应用。

我们将利用链表与堆栈之间的密切关系,通过复用链表类实现堆栈类。我们使用两种不同形式的复用。首先,我们通过链表类的 `private` 继承实现堆栈类,然后通过复合实现一致操作的堆栈类,将链表类作为堆栈类的私有成员。当然,本章的所有数据结构(包括这两个堆栈类)都用模板实现,以提高进一步复用的能力。

图 15.9 的程序(其输出如图 15.10)生成一个 `Stack` 类模板,主要通过 `private` 继承图 15.3 的 `List` 类模板。我们要求 `Stack` 具有成员函数 `push`、`pop`、`isStackEmpty` 和 `printStack`。注意这实际上是 `List` 类模板的 `insertAtFront`、`removeFromFront`、`isEmpty` 和 `print` 函数。当然, `List` 类模板还包含其他成员函数(即 `insertAtBack` 和 `removeFromBack`),这些函数不能通过 `Stack` 类的 `public` 接口访问。因此指明 `Stack` 类模板从 `List` 类模板继承,我们指定 `private` 继承。这样, `List` 的所有成员函数在 `Stack` 类模板中均为 `private`。实现 `Stack` 成员函数时,我们只要进行相应 `List` 成员函数调用即可,即 `push` 调用 `insertAtFront`、`pop` 调用 `removeFromFront`、`isStackEmpty` 调用 `isEmpty` 和 `printStack` 调用 `print`。

```

1 // Fig. 15.9: stack.h
2 // Stack class template definition
3 // Derived from class List
4 #ifndef STACK_H
5 #define STACK_H
6
7 #include "list.h"
8
9 template< class STACKTYPE >
10 class Stack : private List< STACKTYPE > {
11 public:
12     void push( const STACKTYPE &d ) { insertAtFront( d ); }
13     bool pop( STACKTYPE &d ) { return removeFromFront( d ); }
14     bool isStackEmpty() const { return isEmpty(); }
15     void printStack() const { print(); }
16 };
17
18 #endif
19 // Fig. 15.9: fig15_09.cpp
20 // Driver to test the template Stack class
21 #include <iostream.h>
22 #include "stack.h"
23
24 int main()
25 {
26     Stack< int > intStack;
27     int popInteger;
28     cout << "processing an integer Stack" << endl;
29
30     for ( int i = 0; i < 4; i++ ) {

```

```
31     intStack.push( i );
32     intStack.printStack();
33 }
34
35 while ( !intStack.isEmpty() ) {
36     intStack.pop( popInteger );
37     cout << popInteger << " popped from stack" << endl;
38     intStack.printStack();
39 }
40
41 Stack< double > doubleStack;
42 double val = 1.1, popdouble;
43 cout << "processing a double Stack" << endl;
44
45 for ( i = 0; i < 4; i++ ) {
46     doubleStack.push( val );
47     doubleStack.printStack();
48     val += 1.1;
49 }
50
51 while ( !doubleStack.isEmpty() ) {
52     doubleStack.pop( popdouble );
53     cout << popdouble << " popped from stack" << endl;
54     doubleStack.printStack();
55 }
56 return 0;
57 }
```

图 15.9 简单堆栈程序

processing an integer Stack  
The list is: 0

The list is: 1 0

The list is: 2 1 0

The list is: 3 2 1 0

3 popped from stack  
The list is: 2 1 0

2 popped from stack  
The list is: 1 0

1 popped from stack  
The list is: 0

0 popped from stack  
The list is empty

processing a double Stack  
The list is: 1.1

The list is: 2.2 1.1

```

The list is: 3.3 2.2 1.1

The list is: 4.4 3.3 2.2 1.1

4.4 popped from stack
The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1

2.2 popped from stack
The list is: 1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed

```

图 15.10 图 15.9 程序的示例输出

堆栈类模板在main中用于实例化Stack<int>类型的整数堆栈intStack。整数0到3压入intStack中,然后从intStack中弹出。接着用堆栈类模板实例化Stack<double>类型的浮点数堆栈doubleStack,数值1.1、2.2、3.3和4.4压入doubleStack中,然后从doubleStack中弹出。

另一种实现堆栈类模板的方法是通过复合复用List类模板。图15.11的程序用List程序中的文件list.h和listnd.h,并使用与上述Stack程序相同的驱动程序,只是用新的头文件stack\_c.h代替头文件stack.h而已。这两种实现的输出都是一样的。这时堆栈类模板定义包括List<STACKTYPE>类型的成员对象s。

```

1 // Fig. 15.11: stack_c.h
2 // Definition of Stack class composed of List object
3 #ifndef STACK_C
4 #define STACK_C
5 #include "list.h"
6
7 template< class STACKTYPE >
8 class Stack {
9 public:
10     // no constructor; List constructor does initialization
11     void push( const STACKTYPE &d ) { s.insertAtFront( d ); }
12     bool pop( STACKTYPE &d ) { return s.removeFromFront( d ); }
13     bool isEmpty() const { return s.isEmpty(); }
14     void printStack() const { s.print(); }
15 private:
16     List< STACKTYPE > s;
17 };
18
19 #endif

```

图 15.11 使用复合的简单堆栈程序

## 15.6 队列

另一种常见的数据结构是“队列”(queue)。队列类似于购物的队伍,队伍中的第一个人首先得到服务,新来的顾客只能排在队尾等待。队列的节点只能从队头删除,新的节点只能添加在队尾。因此,队列被称为“先进先出”(first-in, first-out; FIFO)的数据结构。插入和删除操作被称为“入队”(enqueue)和“出队”(dequeue)。

队列在计算机系统中有许多应用。大多数计算机只有一个处理器,所以同一个时候只能为一个用户服务。其他等待的用户在队列中登记,每个登记项随着处理器为新用户服务而逐渐向队列的前部靠拢。队列最前面的一个登记项是下一个接收服务的对象。

队列还用于支持打印假脱机。多用户环境只能只有一台打印机,但是可能会有许多用户把输出送到打印机。打印机在忙着的时候仍然会有其他输出要送到打印机,这些输出就会假脱机输出到磁盘,在队列中等待直到打印机可用为止。

在计算机网络中,信息包也是在队列中等待的。每次一个报文到达某个网络结点时,必须把它沿到达最终目标的路径发送到网络上的下一个结点。因为路由结点一次发送一个报文,所以其他的报文要在队列中等待,直到路由器能够发送它们为止。图 15.12 图示了一个多节点队列,注意指向队头的指针和指向队尾的指针。

### 常见编程错误 15.8

没有把队列最后一个节点中的链节设置为空(0)。

图 15.12 的程序(其输出见图 15.13)生成一个 Queue 类模板,主要通过 private 继承图 15.3 的 List 类模板。我们要求 Queue 具有成员函数 enqueue、dequeue、isEmpty 和 printQueue。注意这实际上是 List 类模板的 insertAtBack、removeFromFront、isEmpty 和 print 函数。当然, List 类模板还包含其他成员函数(即 insertAtFront 和 removeFromBack),这些函数不能通过 Queue 类的 public 接口访问。因此指明 Queue 类模板从 list 类模板继承时,我们指定 private 继承。这样, list 的所有成员函数在 Queue 类模板中均为 private。实现 Queue 成员函数时,我们只要进行相应的 List 成员函数调用即可,即 enqueue 调用 insertAtBack、dequeue 调用 removeFromFront、isEmpty 调用 isEmpty 和 printQueue 调用 print。

```
1 // Fig. 15.12: queue.h
2 // Queue class template definition
3 // Derived from class List
4 #ifndef QUEUE_H
5 #define QUEUE_H
6
7 #include "list.h"
8
9 template< class QUEUETYPE >
10 class Queue: private List< QUEUETYPE > {
11 public:
12     void enqueue( const QUEUETYPE &d ) { insertAtBack( d ); }
13     bool dequeue( QUEUETYPE &d )
14         { return removeFromFront( d ); }
15     bool isEmpty() const { return isEmpty(); }
16     void printQueue() const { print(); }
17 };
```

```

18
19 #endi
20 // Fig. 15.12: fig15_12.cpp
21 // Driver to test the template Queue class
22 #include <iostream.h>
23 #include "queue.h"
24
25 int main()
26 {
27     Queue< int > intQueue;
28     int dequeueInteger;
29     cout << "processing an integer Queue" << endl;
30
31     for ( int i = 0; i < 4; i++ ) {
32         intQueue.enqueue( i );
33         intQueue.printQueue();
34     }
35
36     while ( !intQueue.isQueueEmpty() ) {
37         intQueue.dequeue( dequeueInteger );
38         cout << dequeueInteger << " dequeued" << endl;
39         intQueue.printQueue();
40     }
41
42     Queue< double > doubleQueue;
43     double val = 1.1, dequeuedouble;
44
45     cout << "processing a double Queue" << endl;
46
47     for ( i = 0; i < 4; i++ ) {
48         doubleQueue.enqueue( val );
49         doubleQueue.printQueue();
50         val += 1.1;
51     }
52
53     while ( !doubleQueue.isQueueEmpty() ) {
54         doubleQueue.dequeue( dequeuedouble );
55         cout << dequeuedouble << " dequeued" << endl;
56         doubleQueue.printQueue();
57     }
58
59     return 0;
60 }

```

图 15.12 处理队列

队列类模板在 main 中用于实例化 Queue<int> 类型的整数队列 intQueue。整数 0 到 3 放进队列 intQueue，然后按先进先出的顺序出队。接着用队列类模板实例化 Queue<double> 类型的浮点数队列 doubleQueue，数值 1.1、2.2、3.3 和 4.4 放进队列 doubleQueue，然后按先进先出的顺序出队。

```

Processing an integer queue
The list is: 0

```

```

The list is: 0 1

```

```
The list is: 0 1 2

The list is: 0 1 2 3

0 dequeued
The list is: 1 2 3

1 dequeued
The list is: 2 3

2 dequeued
The list is: 3
3 dequeued
The list is empty

Processing a float Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

The list is: 1.1 2.2 3.3 4.4

1.1 dequeued
The list is: 2.2 3.3 4.4

2.2 dequeued
The list is: 3.3 4.4

3.3 dequeued
The list is: 4.4

4.4 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

图 15.13 图 15.12 程序的示例输出

## 15.7 树

链表、堆栈和队列都是线性数据结构。树是一种具有某些特定属性的非线性的二维数据结构。树的节点包含两个或多个链节。本节讨论二叉树(图 15.14), 即所有节点都只包含两个链节的树(链节可以是空)。根节点是树中的第一个节点, 根节点中的每一个链节都引用一个子节点。左边的子节点是左子树的第一个节点, 右边的子节点是右子树的第一个节点。一个节点的两个子节点称为“兄弟节点”。没有子节点的节点称为“叶节点”。和自然界的树的形式相反, 计算机科学家通常从根节点开始绘制树。

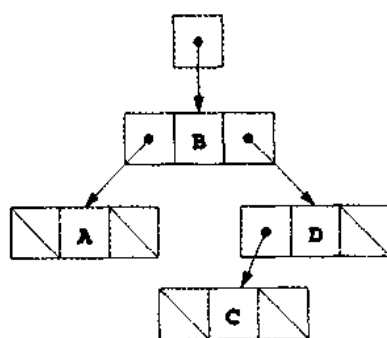


图 15.14 二叉树的图示

本节要建立一种特别的二叉树称为“二叉查找树”。二叉查找树(没有重复的节点值)的特点是:左子树上的所有的值都小于其父节点的值,而所有右子树上的值都大于其父节点上的值。图 15.15 表示了一个有 12 个值的二叉查找树。对应于一组数据的二叉树的形状能够随这些值插入到二叉树中的顺序而变化。

#### 常见编程错误 15.9

树的叶节点中的链节没有设置为空(0)。

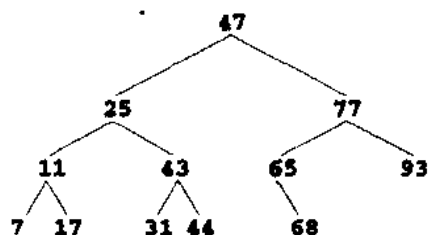


图 15.15 二叉查找树

图 15.16 中的程序(输出见图 15.17)建立了一个二叉树,并以中序遍历、前序遍历和后序遍历三种方法遍历该二叉树。

函数 main 首先实例化 `Tree<int>` 类型的整数树 `intTree`。程序提示输入 10 个整数,通过 `insertNode` 插入二叉树中,然后程序对 `intTree` 进行前序遍历、中序遍历和后序遍历(稍后解释)。然后,程序实例化 `Tree<double>` 类型的整数树 `doubleTree`。程序提示输入 10 个双精度的值,通过 `insertNode` 插入二叉树中,然后程序对 `doubleTree` 进行前序遍历、中序遍历和后序遍历。

下面要介绍类模板定义。首先 `TreeNode` 类模板声明 `Tree` 类模板为其友元。`TreeNode` 类的 `private` 数据取节点的 `data` 值,并有指针 `leftPtr` (节点左子树的指针)和 `rightPtr` (节点右子树的指针)。构造函数将 `data` 设置为构造函数参数中提供的值,并将 `leftPtr` 和 `rightPtr` 设置为 0 (从而将这个节点初始化为叶节点)。成员函数 `getData` 返回 `data` 值。

`Tree` 类有一个 `private` 数据 `rootPtr`, 是树的根节点指针。类包括 `public` 成员函数 `insertNode` (在树中插入新节点)、`preorderTraversal`、`inorderTraversal` 和 `postorderTraversal` (分别对树进行前序、中序和后序遍历)。这些成员函数调用自己的递归工具函数对树的内部表示进行相应操作。`Tree` 构造函数将 `rootPtr` 初始化为 0, 表示树最初为空。

```
1 // Fig. 15.16: treenode.h
2 // Definition of class TreeNode
3 #ifndef TREENODE_H
4 #define TREENODE_H
5
6 template< class NODETYPE > class Tree; // forward declaration
7
8 template< class NODETYPE >
9 class TreeNode {
10     friend class Tree< NODETYPE >;
11 public:
12     TreeNode( const NODETYPE &d )
13         : leftPtr( 0 ), data( d ), rightPtr( 0 ) { }
14     NODETYPE getData() const { return data; }
15 private:
16     TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
17     NODETYPE data;
18     TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
19 };
20
21 #endif
22 // Fig. 15.16: fig15_16.cpp
23 // Definition of template class Tree
24 #ifndef TREE_H
25 #define TREE_H
26
27 #include <iostream.h>
28 #include <assert.h>
29 #include "treenode.h"
30
31 template< class NODETYPE >
32 class Tree {
33 public:
34     Tree();
35     void insertNode( const NODETYPE & );
36     void preOrderTraversal() const;
37     void inOrderTraversal() const;
38     void postOrderTraversal() const;
39 private:
40     TreeNode< NODETYPE > *rootPtr;
41
42     // utility functions
43     void insertNodeHelper(
44         TreeNode< NODETYPE > **, const NODETYPE & );
45     void preOrderHelper( TreeNode< NODETYPE > * ) const;
46     void inOrderHelper( TreeNode< NODETYPE > * ) const;
47     void postOrderHelper( TreeNode< NODETYPE > * ) const;
48 };
49
50 template< class NODETYPE >
51 Tree< NODETYPE >::Tree() { rootPtr = 0; }
52
53 template< class NODETYPE >
54 void Tree< NODETYPE >::insertNode( const NODETYPE &value )
55     { insertNodeHelper( &rootPtr, value ); }
```



```
56
57 // This function receives a pointer to a pointer so the
58 // pointer can be modified.
59 template< class NODETYPE >
60 void Tree< NODETYPE >::insertNodeHelper(
61     TreeNode< NODETYPE > **ptr, const NODETYPE &value )
62 {
63     if ( *ptr == 0 ) {                // tree is empty
64         *ptr = new TreeNode< NODETYPE >( value );
65         assert( *ptr != 0 );
66     }
67     else                               // tree is not empty
68         if ( value < ( *ptr )->data )
69             insertNodeHelper( &( ( *ptr )->leftPtr ), value );
70         else
71             if ( value > ( *ptr )->data )
72                 insertNodeHelper( &( ( *ptr )->rightPtr ), value );
73             else
74                 cout << value << " dup" << endl;
75 }
76
77 template< class NODETYPE >
78 void Tree< NODETYPE >::preOrderTraversal() const
79 { preOrderHelper( rootPtr ); }
80
81 template< class NODETYPE >
82 void Tree< NODETYPE >::preOrderHelper(
83     TreeNode< NODETYPE > *ptr ) const
84 {
85     if ( ptr != 0 ) {
86         cout << ptr->data << ' ';
87         preOrderHelper( ptr->leftPtr );
88         preOrderHelper( ptr->rightPtr );
89     }
90 }
91
92 template< class NODETYPE >
93 void Tree< NODETYPE >::inOrderTraversal() const
94 { inOrderHelper( rootPtr ); }
95
96 template< class NODETYPE >
97 void Tree< NODETYPE >::inOrderHelper(
98     TreeNode< NODETYPE > *ptr ) const
99 {
100     if ( ptr != 0 ) {
101         inOrderHelper( ptr->leftPtr );
102         cout << ptr->data << ' ';
103         inOrderHelper( ptr->rightPtr );
104     }
105 }
106
107 template< class NODETYPE >
108 void Tree< NODETYPE >::postOrderTraversal() const
109 { postOrderHelper( rootPtr ); }
110
111 template< class NODETYPE >
```

```
112 void Tree< NODETYPE >::postOrderHelper(  
113     TreeNode< NODETYPE > *ptr ) const  
114 {  
115     if ( ptr != 0 ) {  
116         postOrderHelper( ptr->leftPtr );  
117         postOrderHelper( ptr->rightPtr );  
118         cout << ptr->data << ' ';  
119     }  
120 }  
121  
122 #endif  
123 // Fig. 15.16: fig15_16.cpp  
124 // Driver to test class Tree  
125 #include <iostream.h>  
126 #include <iomanip.h>  
127 #include "tree.h"  
128  
129 int main()  
130 {  
131     Tree< int > intTree;  
132     int intVal;  
133  
134     cout << "Enter 10 integer values:\n";  
135     for( int i = 0; i < 10; i++ ) {  
136         cin >> intVal;  
137         intTree.insertNode( intVal );  
138     }  
139  
140     cout << "\nPreorder traversal\n";  
141     intTree.preOrderTraversal();  
142  
143     cout << "\nInorder traversal\n";  
144     intTree.inOrderTraversal();  
145  
146     cout << "\nPostorder traversal\n";  
147     intTree.postOrderTraversal();  
148  
149     Tree< double > doubleTree;  
150     double doubleVal;  
151  
152     cout << "\n\nEnter 10 double values:\n"  
153         << setiosflags( ios::fixed | ios::showpoint )  
154         << setprecision( 1 );  
155     for ( i = 0; i < 10; i++ ) {  
156         cin >> doubleVal;  
157         doubleTree.insertNode( doubleVal );  
158     }  
159  
160     cout << "\nPreorder traversal\n";  
161     doubleTree.preOrderTraversal();  
162  
163     cout << "\nInorder traversal\n";  
164     doubleTree.inOrderTraversal();  
165  
166     cout << "\nPostorder traversal\n";
```

```

167     doubleTree.postOrderTraversal();
168
169     return 0;
170 }

```

图 15.16 生成与遍历二叉树

Tree类的工具函数insertNodeHelper在树中递归插入节点。二叉查找树中的节点只能作为叶节点插入。如果树是空的，则生成并初始化新的TreeNode插入树中。

如果树不是空的，则程序比较要插入的值与根节点中的data值。如果插入的值更小，则程序递归调用insertNodeHelper，将该值插入左子树中。如果插入的值更大，则程序递归调用insertNodeHelper，将该值插入右子树中。如果插入的值等于根节点中的data值，则程序打印消息“dup”并返回，不把重复值插入树中。成员函数inOrderTraversal、preOrderTraversal和postOrderTraversal遍历该树（见图15.18）并打印节点值。

```

Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50

Enter 10 double values:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5
Inorder traversal
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2

```

图 15.17 图 15.16 程序的示例输出

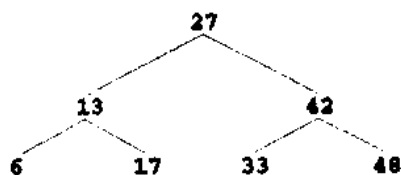


图 15.18 二叉查找树

inOrderTraversal（中序遍历）的步骤如下所示：

1. 用 inOrderTraversal 遍历左子树。
2. 处理节点中的值（即打印节点值）。
3. 用 inOrderTraversal 遍历右子树。

节点中的值是在处理完其左子树中的值后才处理的。图 15.18 中的树的中序遍历为：

6 13 17 27 33 42 48

注意 `inOrderTraversal` 遍历二叉查找树以升序方式打印了节点的值。建立二叉查找树的过程实际上是数据排序的过程，因此把这个过程称为“二叉树排序”（binary tree sort）。

`preOrderTraversal`（前序遍历）遍历的步骤如下所示：

1. 处理节点中的值。
2. 用 `preOrderTraversal` 遍历左子树。
3. 用 `preOrderTraversal` 遍历右子树。

每一个节点中的值是在访问该节点时处理的。在处理完给定节点中的值后，处理左子树中的值，然后处理右子树中的值。图 15.18 中的树的前序遍历为：

27 13 6 17 42 33 48

`postOrderTraversal`（后序遍历）的步骤如下所示：

1. 用 `postOrderTraversal` 遍历左子树。
2. 用 `postOrderTraversal` 遍历右子树。
3. 处理节点中的值。

每一个节点中的值是在打印出其子节点中的值后打印的。图 15.18 中的树的后序遍历为：

6 17 13 33 48 42 27

二叉查找树可容易地去除重复的值。在建立一个树的时候，如果试图插入一个重复的值就会判断出来，因为在判断重复的值的“左”“右”去向时，它既不能插入到左边也不能插入到右边。这时程序会把重复的值删除。

在二叉树中查找与某个关键字匹配的值也是很快的。如果一棵树是满二叉树，那么每一层包含的元素个数都是上一层的两倍。因此，具有  $n$  个元素的二叉树，最多有  $\log_2 n$  层，所以最多比较  $\log_2 n$  次就可以找到匹配值或确定不存在匹配值。例如，查找有 1000 个元素的二叉树，比较次数不超过 10 次（因为  $2^{10} > 1000$ ）；查找有 1 000 000 个元素的二叉树，比较次数不超过 20 次（因为  $2^{20} > 1\,000\,000$ ）。

练习中介绍了操作二叉树的其他几种算法，如删除二叉树中的某项、以二维树格式打印二叉树和按二叉树的层次顺序遍历二叉树。按二叉树的层次顺序遍历二叉树是从根节点开始逐行访问节点。在二叉树的每一层中，节点按从左到右的顺序访问。其他有关二叉树的练习包括允许查找包含重复值的二叉树、在二叉树中插入字符串值以及确定二叉树有多少层。

## 小结

- 自引用结构包含了称为“链节”的成员，链节是一个指针，它指向类型相同的结构。
- 自引用结构能够以堆栈、队列、链表和树的形式把许多结构链接在一起。
- 动态内存分配的内存中保留了在程序执行过程中存储数据对象的字节块。
- 链表是自引用类对象的线性集合。
- 链表是一种动态数据结构，其长度可在需要的时候增大和缩小。

- 只要有可用的内存, 链表可不断地增大。
- 通过指针操作, 链表提供了一种简单的插入和删除数据的机制。
- 单向链表以第一个节点的指针开始, 每个节点包含下一个顺序节点的指针。这个链表在节点的指针成员为0时终止。单向链表只能单向遍历。
- 循环单向链表以第一个节点的指针开始, 每个节点包含下一个顺序节点的指针, 最后一个节点不是包含0指针, 而是指回第一个节点, 从而形成循环。
- 双向链表可以正向和逆向遍历。每个节点都有一个正向指针, 指向往前的一个节点, 还有一个逆向指针, 指向往后的一个节点。
- 在循环双向链表中, 最后一个节点的正向指针指向第一个节点, 第一个节点的逆向指针指向最后一个节点, 从而形成循环。
- 堆栈和队列是一种特殊的链表。
- 堆栈节点的插入和删除在栈顶进行, 因此堆栈被称为后进先出的数据结构。
- 堆栈的最后一个节点的链节成员要设置为表示栈底的空(0)。
- “压入”(push)和“弹出”(pop)是堆栈操作的两种主要形式。压入操作在栈顶建立了一个新的节点。弹出操作删除栈顶节点、释放分配给弹出节点的内存、返回弹出的值。
- 在队列数据结构中, 节点的删除和插入分别在队头和队尾进行。因此, 队列被称为先进先出的数据结构。插入和删除操作被称为“入队”(enqueue)和“出队”(dequeue)。
- 树是二维数据结构, 每个节点需要有两个或多个链节。
- 二叉树的每个节点包含两个链节。
- 根节点是二叉树的第一个节点。
- 根节点中的每一个指针都引用一个子节点。左边的子节点是左子树的第一个节点, 右边的子节点是右子树的第一个节点。一个节点的子节点称为“兄弟节点”。没有子节点的节点称为叶节点。
- 二叉查找树的特点是: 左子树上的所有的值都小于其父节点的值, 而所有右子树上的值都大于其父节点上的值。如果能够确定没有重复的数据值, 那么右子节点中的值大于父节点中的值。
- 二叉树的中序遍历次序是: 中序遍历左子树、处理节点中的值、中序遍历右子树。节点中的值是在处理完其左子树中的值后处理的。
- 前序遍历次序是: 处理节点中的值、前序遍历左子树、前序遍历右子树。每一个节点中的值是在访问该节点时处理的。
- 后序遍历次序是: 后序遍历左子树、后序遍历右子树、处理节点中的值。每一个节点中的值是在处理完其子树中的值后处理的。

## 术语

binary search tree 二叉查找树

binary tree 二叉树

binary tree sort 二叉树排序

child node 子节点

children 子节点

circular, doubly-linked list 循环双向链表

circular, singly-linked list 循环单向链表

deleting a node 删除节点

dequeue

double indirection 双间接

doubly-linked list 双向链表

duplicate elimination 重复消除

dynamic data structures 动态数据结构	遍历
enqueue	dynamic memory allocation 动态内存分配
FIFO (first-in, first-out) FIFO (先进先出)	predicate function 判定函数
head of a queue 队头	preorder traversal of a binary tree 二叉树前序遍历
inorder traversal of a binary tree 二叉树中序遍历	push
inserting a node 插入一个节点	queue 队列
leaf node 叶节点	right child 右子节点
left child 左子节点	right subtree 右子树
left subtree 左子树	root node 根节点
level-order traversal of a binary tree 二叉树的层次顺序遍历	self-referential structure 自引用结构
LIFO (last-in, first-out) LIFO (后进先出)	siblings 兄弟节点
linear data structure 线性数据结构	singly-linked list 单向链表
linked list 链表	sizeof
node 节点	stack 堆栈
nonlinear data structure 非线性数据结构	subtree 子树
null pointer null 指针	tail of a queue 队尾
parent node 父节点	top 顶
pointer to a pointer 指向指针的指针	traversal 遍历
pop	tree 树
postorder traversal of a binary tree 二叉树后序遍历	visit a node 访问节点

## 自测练习

### 15.1 填空:

- 自 \_\_\_\_\_ 结构用来构造动态数据结构, 可以在执行时伸缩。
- 运算符 \_\_\_\_\_ 用来动态分配内存, 这个运算符返回所分配内存的指针。
- \_\_\_\_\_ 是一种特殊的链表, 它只允许在链表的开始位置插入和删除节点, 按后进先出顺序返回节点值。
- 不对链表做任何修改而只是判断链表是否为空的函数称为 \_\_\_\_\_ 函数。
- 队列中先插入的节点也是先要被删除节点, 所以队列被称为 \_\_\_\_\_ 数据结构。
- 指向链表下一个节点的指针称为 \_\_\_\_\_。
- 运算符 \_\_\_\_\_ 用来回收动态分配的内存。
- \_\_\_\_\_ 是一种特殊的链表, 节点只能插入到这种链表的头部, 删除只能发生在这种链表的尾部。
- \_\_\_\_\_ 是一种非线性的二维数据结构, 它的节点包含两个或多个链节。
- 堆栈中最后插入的节点也是先要被删除的节点, 所以堆栈被称为 \_\_\_\_\_ 数据结构。
- \_\_\_\_\_ 树的节点包含两个链节成员。
- 树的第一个节点是 \_\_\_\_\_ 节点。
- 树节点中的每一个节点指向该节点的 \_\_\_\_\_ 或 \_\_\_\_\_。
- 没有子节点的树节点称为 \_\_\_\_\_ 节点。

- o) 遍历二叉树的三种算法是 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- 15.2 链表和堆栈有什么区别?
- 15.3 堆栈和队列有什么区别?
- 15.4 本章的标题也许可以改变成“可复用数据结构”。请说明下列概念和术语对可复用数据结构的作用:
- 类
  - 模板类
  - 继承
  - 私有继承
  - 复合
- 15.5 写出图 15.19 中二叉树的中序遍历、前序遍历和后序遍历情况。

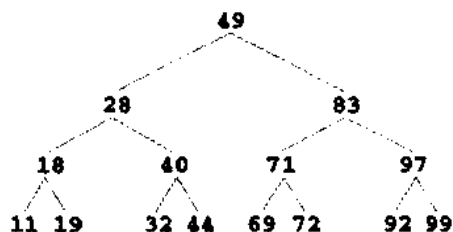


图 15.19 有 15 个节点的二叉树

### 自测练习答案

- 15.1 a) 引用。b) new。c) 堆栈。d) 判定。e) FIFO。f) 链节。g) delete。h) 队列。i) 树。j) LIFO。k) 二叉。l) 根。m) 子节点、子树。n) 叶。o) 中序遍历、前序遍历、后序遍历。
- 15.2 链表允许把节点插入到任何位置和删除任何位置的节点。但是，堆栈只允许节点在栈顶插入和删除。
- 15.3 队列拥有指向队头和队尾的两个指针，因此节点的插入和删除分别是在队尾和队头进行的。堆栈只有指向栈顶的指针，因此插入和删除都在栈顶进行。
- 15.4 a) 类可以将某种类型（即类）实例化为多个数据结构对象。
- b) 类模板可以实例化相关类（根据不同类型参数），然后对每个模板类生成多个对象。
- c) 继承可以在派生类中复用基类代码，因此派生类数据结构也是基类数据结构（对于公有继承）。
- d) 私有继承可以复用基类的部分代码形成派生类数据结构，由于是私有继承，基类中的所有公有成员函数在派生类中变成私有。这样就可以防止派生类数据结构的客户访问不适用于派生类的基类成员函数。
- e) 复合可以让类对象数据结构变成复合类的成员，从而复用代码。如果类对象变为复合类的私有成员，则类对象的公有成员函数也无法通过复合对象接口访问。
- 15.5 中序遍历为：
- 11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
- 前序遍历为：

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

后序遍历为:

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

## 练习

- 15.6 编写一个程序,把两个字符链表连接起来。程序包含函数 `concatenate`, 函数以对两个链表的引用作为参数,把第二个链表连接到第一个链表上。
- 15.7 编写一个程序,把两个有序排列的整数链表合并成一个有序排列的整数链表。函数 `merage` 以两个指针作为参数,这两个指针分别指向两个待合并链表的头节点,函数返回一个指针,它指向合并后的链表的头节点。
- 15.8 编写一个程序,向链表中按顺序插入 0 到 100 之间的 25 个随机整数。程序要计算出插入的所有元素的总和及平均值(用浮点数表示)。
- 15.9 编写一个程序,建立一个包含 10 个字符的链表,再建立一个新链表,包含第一个链表的反序副本。
- 15.10 编写一个程序,输入一行文本再用堆栈反序打印文本。
- 15.11 编写一个程序,用堆栈确定某个字符串是否回文(即正读和反读相同),忽略空格和标点符号。
- 15.12 编译器经常用堆栈计算表达式和产生机器语言代码。在该练习和下一个练习中,我们要考察编译器是怎样计算由常量、运算符和括号组成的表达式。

人们在写表达式(如  $3 + 4$ 、 $7/9$ )时,总是把运算符(这里是  $+$  和  $/$ )放在操作数之间(称为中缀表达式法)。计算机更喜欢使用后缀表示法,也就是把运算符写在两个操作数的右边。前面的中缀表达式用后缀表示法中分别表示为  $34 +$  和  $79/$ 。

在计算一个复杂的中缀表达式时,编译器首先把它转化成后缀表示法,然后再计算这个后缀表达式。所有算法都只需要对表达式做一次从右到左的扫描,每个算法都用堆栈支持它的操作,堆栈在每个算法中的用途不同。

这个练习要求用 C++ 语言编写一个把中缀转化成后缀的算法,下一个练习要求用 C++ 语言编写一个计算后缀表达式的求值算法。在本章的后面读者就会发现,本练习中编写的代码有助于实现完全运行的编译器。

编写一个程序,把由单个数字的整数组成的中缀代数表达式(假定键入的是一个正确的表达式),例如:

$(6 + 2) * 5 - 8 / 4$

转化成一个后缀表达式。上面这个中缀表达式的后缀形式为:

$6 2 + 5 * 8 4 -$

程序把表达式读入字符数组 `infix`, 并用在本章实现的改进的堆栈函数在字符数组 `postfix` 中建立后缀表达式。建立后缀表达式的算法如下所示:

- 1) 把左括号 '(' 压入堆栈。
- 2) 在 `infix` 的末端补右括号 ')'。



3) 若堆栈非空, 从左到右读取 infix 并做下列工作:

如果 infix 中的当前字符是一个数字, 把它复制给 postfix 中的下一个元素。

如果 infix 中的当前字符是左括号, 把它压入堆栈。

如果 infix 中的当前字符是运算符:

若栈顶运算符的优先级等于或高于当前运算符, 将其弹出堆栈并插入到 postfix 中, 把当前字符压入堆栈。

如果 infix 中的当前字符是右括号:

从栈顶弹出运算符并把它们插入到 postfix 中直到左括号位于栈顶, 从堆栈中弹出 (并删除) 左括号。

表达式中可以使用下列运算符:

+	加
-	减
*	乘
/	除
^	幂
%	求余

堆栈要维护堆栈节点, 各节点包含数据成员和下一个堆栈节点的指针。

可能要提供下列功能:

- 函数 convertToPostfix 把中缀表达式转化成后缀表达式。
- 函数 isOperator 判断 c 是否是一个运算符。
- 函数 precedence 判断 operator1 的优先级是低于、等于还是高于 operator2 的优先级, 对应的函数返回值分别是 -1、0 和 1。
- 函数 push 把一个值压入堆栈。
- 函数 pop 从堆栈中弹出一个值。
- 函数 stackTop 返回栈顶的值但不把这个值从堆栈内弹出。
- 函数 isEmpty 判断堆栈是否为空。
- 函数 printStack 打印堆栈。

15.13 编写一个程序, 计算一个后缀表达式 (假定它是合法的) 的值, 例如:

6 2 + 5 \* 8 4 / -

程序把一个由数字和运算符组成的后缀表达式读入到一个数组中, 用本章开始部分实现的改进的堆栈函数扫描并计算这个表达式。算法如下:

- 1) 在后缀表达式的末端补上空字符 ('0'), 在下面的处理中遇到空字符时停止进一步处理。
- 2) 如没有遇到 '0', 从左到右读取表达式:
  - 如果当前字符是一个数字,
  - 将其整数值压入堆栈 (一个数字字符的整数值等于它在计算机字符集中的值减去字符 '0' 在计算机字符集中的值)。
  - 否则, 如果当前字符集是一个运算符, 弹出栈顶的两个元素并赋给变量 x 和 y。

计算  $y \text{ operator } x$  的值。

把计算结果压入堆栈。

3) 当在表达式中遇到空字符时, 弹出栈顶值。这就是后缀表达式的结果。

注意: 在上面的 2) 中, 若运算符是 '/', 栈顶为 2, 栈中的下一个元素是 8, 那么弹出 2 并赋给  $x$ , 计算  $8/2$ , 把结果 4 压回堆栈。这同样适用于运算符 '-'。表达式中可使用的算术运算符有:

+	加
-	减
*	乘
/	除
^	幂
%	求模

堆栈要维护堆栈节点, 各节点包含 `int` 数据成员和下一个堆栈节点的指针。

可能要提供下列功能:

- a) 函数 `evaluatePostfixExpression` 计算后缀表达式的值。
- b) 函数 `calculate` 计算 `op1 operator op2` 的值。
- c) 函数 `push` 把一个值压入堆栈。
- d) 函数 `pop` 从堆栈中弹出一个值。
- e) 函数 `isEmpty` 判断堆栈是否为空。
- f) 函数 `printStack` 打印堆栈。

15.14 修改练习 15.13 中计算后缀表达式的程序, 使它能够处理大于 9 的操作数。

15.15 (超市模拟) 编写一个程序, 模拟在超市排队结账的队伍, 这个队伍用队列表示。顾客入队的时间间隔是 1 到 4 分钟之间的一个随机整数, 每位顾客接收服务的时间也是 1 到 4 分钟之间的一个随机整数。显然出队、入队的速度应该平衡好, 如果入队的平均速度高于出队的平均速度, 那么队列的长度就会无限的增加, 即使速度平衡了, 由于随机性也会造成很长的队列。用下列算法运行超市模拟程序 12 个小时 (720 分钟):

1) 选出一个 1 到 4 之间的随机整数, 以确定第一位顾客在第几分钟入队。

2) 在第一位顾客入队时:

    确定顾客接受服务的时间 (1 到 4 之间的随机整数);

    开始为顾客服务;

    计划下一位顾客的到达时刻 (1 到 4 之间的随机整数加上当前时刻)。

3) 这天的每一分钟:

    若下一名顾客到达, 让这位顾客入队, 计划下一位顾客到达的时刻;

    如果对上一位顾客的服务完毕, 让下一位要接受服务的顾客出队, 确定顾客的服务完成时刻 (1 到 4 之间的一个随机整数加上当前时刻)。

运行模拟程序 720 分钟, 回答下列问题:

- a) 队列中最多的顾客数目是多少?
- b) 顾客等待的最长时间是多少?
- c) 如果到达队列的间隔由 1 到 4 分钟改为 1 到 3 分钟, 结果怎样?

- 15.16 修改图 15.16 中的程序,使二叉树能够存放相同的值。
- 15.17 在图 15.16 中的程序的基础上编写一个程序,读取一行文本,把句子打断成独立的单词(用库函数 `strtok`)再将各个单词插入到二叉树中,然后分别打印这棵二叉树的中序、前序和后序遍历结果。用 OOP 方法。
- 15.18 在本章中,我们看到在创建二叉查找树时,去除重复值是很直接的:试描述怎样只用一个一维数组去除重复值,比较基于数组和基于二叉查找树去除重复值的性能。
- 15.19 编写函数 `depth`,它以一个二叉树作为参数,确定这棵二叉树有多少层。
- 15.20 (用递归打印反序链表)编写函数 `printlistBackwards`,用反序递归输出链表中的数据项。编写一个使用该函数的测试程序,建立一个有序整数链表,并把这个链表反序打印出来。
- 15.21 (用递归查找链表)编写函数 `searchList`,用递归方法在链表中查找某个指定的值。如果找到了这个值,函数返回指向这个值的指针,否则返回空。编写一个使用该函数的测试程序,在程序中建立一个整数链表,并提示用户键入要查找的值。
- 15.22 (删除二叉树的节点)这个练习讨论了从二叉树中删除数据项。删除算法不像插入算法那样直接,在删除一个数据项时会遇到三种情况:数据项存放在叶节点中(即它没有子节点)、数据项存放的节点有一个子节点、数据项存放的节点有两个子节点。

如果要删除存放在叶节点中的数据项,删除这个节点再把父节点中的指针设置为空。

如果存放要删除的数据项的节点有一个子节点,把父节点的指针指向这个子节点再删除该数据项。这样,子节点在树中取代删除掉的节点。

最后一种情况是最难的,当要删除的节点有两个子节点的时候,必须有另一个节点取代这个节点。但是,不能把父节点中的指针简单地指向要被删除的节点的某个子节点,因为在多数情况下,这样得到的二叉树往往不再符合二叉查找树的特征(即左子树中的任何数据都小于父节点的值,右子树中的任何数据都大于父节点的值)。

用哪个节点作为替换节点才能维持上述特征呢?或者用小于删除节点的树中最大值节点,或者用大于删除节点的树中的最小值节点。我们考虑值较小的那个节点。在二叉查找树中,小于父值的最大值位于父节点的左子树中并且肯定存放在该左子树的最右端节点中,沿着该左子树往下向右走,当到达的节点指向右子节点的指针为空时,这个节点就是要找的节点。将指针指向这个替换节点,该节点或者是一个叶节点或者只有一个左子节点。如果替换节点是一个叶节点,执行删除的步骤如下所示:

- 1) 把指向要删除节点的指针存入一个临时指针变量(这个指针用于动态释放分配的内存)。
- 2) 使要删除节点的父节点指向左子树的指针指向替换节点。
- 3) 把替换节点的父节点指向右子树的指针设置为空。
- 4) 使替换节点的指向右子树的指针指向要删除节点的右子树。
- 5) 使替换节点的指向左子树的指针指向要删除节点的左子树。
- 6) 删除临时指针变量指向的那个节点。

替换节点带有左子节点时的删除步骤与替换节点无子节点时的情况相似,但在算法中必须把子节点移到替换节点的位置。如果替换节点带有左子节点,执行删除操作的步骤如下:

- 1) 把指向要删除节点的指针存入一个临时指针变量中。
- 2) 使要删除节点的父节点指向左子树的指针指向替换节点。
- 3) 使替换节点的父节点指向右子树的指针指向替换节点的左子节点。
- 4) 使替换节点中指向右子树的指针指向要删除节点的右子树。
- 5) 使替换节点的指向左子树的指针指向要删除节点的左子树。
- 6) 删除临时指针变量指向的那个节点。

编写函数 `deleteNode`，它的参数为指向树中根节点的指针和要删除的值。函数先在树中找出存放这个删除值的节点，然后再用上面讨论的方法删除这个节点。如果在树中找不到要删除的值，函数应该打印消息。修改图 15.16 中的程序，使它使用上面这个函数。每删除一个数据项后，调用遍历函数 `inOrder`、`preOrder` 和 `postOrder` 以证明删除操作的正确性。

- 15.23 (二叉树查找) 编写成员函数 `binaryTreeSearch`，查找某个指定的值在二叉树中的位置。函数的参数是指向二叉树根节点的指针和要查找的关键字，如果找到存放这个关键字的节点，函数返回一个指向该节点的指针，否则返回一个空指针。
- 15.24 (按层次顺序遍历二叉树) 图 15.16 中的程序给出了三种遍历二叉树的递归函数——中序遍历、前序遍历和后序遍历。这个练习给出的是二叉树的层次遍历法，所有节点值从根节点开始逐层打印出来，每层的节点是从左到右打印的。层次遍历不是递归算法，它使用队列数据结构控制节点的输出。算法如下：

- 1) 把根节点插入队列。
- 2) 当队列中还有节点时：
  - 取出队列中的下一个节点，打印节点的值；
  - 如果节点中指向左子节点的指针不是空，把左子节点插入队列；
  - 如果节点中指向右子节点的指针不是空，把右子节点插入队列。

编写函数 `levelOrder`，执行一棵二叉树的层次遍历。修改图 15.16 中的程序使它使用这个函数。(注意：这个程序还需要修改和使用图 15.12 中的队列处理函数)。

- 15.25 (打印树) 编写递归函数 `outputTree`，在屏幕上显示一棵二叉树。函数将树逐行输出，树的顶部显示在屏幕的左边，树的底部显示在右边，每行以垂直方向输出。例如，图 15.19 描述的二叉树的输出结果如下所示：

```

          97          99
        83          92
      71          72
    49          69
      40          44
    28          32
      18          19
          11
  
```

注意, 二叉树最右端的叶节点输出在最右列的顶部, 根节点则输出在最左边。每个输出列都是从前一个输出列的右边第五个空格开始的。函数 `outputTree` 的参数是表示在输出结果之前有多少个空格的整数变量 `totalSpaces` (这个变量应该从 0 开始, 以便使根节点输出在屏幕的左边)。函数使用修改后的中序遍历输出二叉树 (从树的最右端节点开始向左推进), 算法如下:

当指向当前节点的指针不等于 NULL 时:

- 用当前节点的右子树和 `totalSpaces + 5` 递归调用 `outputTree`;
- 用 for 结构输出空格, for 结构从 1 到 `totalSpaces` 计数;
- 输出当前节点中的值;
- 使指向当前节点的指针指向当前节点的左子树;
- 将 `totalSpaces` 加 5。

### 特殊小节: 建立自己的编译器

练习 5.18 和 5.19 介绍了 Simpletron 机器语言 (SML), 并建立了 Simpletron 计算机模拟器 (运行用 SML 语言编写的程序)。在这个小节中, 我们要建立一个编译器, 把用高级语言编写的程序转换成用 SML 语言编写的程序。小节中的内容与整个程序设计过程联系紧密。我们要用新的高级语言编写程序、在建好的编译器上编译这种程序并把该程序在练习 5.19 中建立的模拟器上运行。(使用面向对象的方法实现编译器。)

15.26 (Simple 语言) 在我们开始创建一个编译器之前, 先来讨论一个简单的、但是功能强大的高级语言, 它有点类似于 BASIC 的早期版本, 我们把这种语言叫做 Simple 语言。每一条 Simple 语句都包含一个行号和一条 Simple 语言的指令。行号在程序中必须是递增的。每一条指令都是从下面某一个 Simple 命令开始的: `rem`、`input`、`let`、`print`、`goto`、`if/goto`、`end` (见图 15.20), 除 `end` 外, 所有命令都可以重复使用。Simple 只支持含有 `+`、`-`、`*`、`/` 运算符的整数表达式。这些运算符有着和 C 语言中一样的优先级, 括号可以改变表达式的计算顺序。

Simple 编译器只识别小写字母, Simple 文件中的字符都应是小写的 (大写字母会导致语法错误, 但在 `rem` 语句中例外)。一个变量就是一个单字符。Simple 不支持描述性的变量名, 所以应该在注释中指出它们在程序中的用途。Simple 只使用整型变量。Simple 没有变量声明, 只要在程序使用变量名就算是做了声明并自动初始化为零。Simple 语法不支持字符串操作 (即字符串的读、写和比较等等)。如果在 Simple 程序中遇到一个字符串 (在命令而不是 `rem` 之后), 编译器就会产生一条语法错误。第一个版本的编译器总是假定输入的 Simple 程序都是正确的。练习 15.29 要求修改编译程序, 使它能够检查语法错误。

Simple 用 `if/goto` 条件语句和 `goto` 无条件语句改变程序的流程。如果 `if/goto` 语句中的条件成立, 控制就转移到程序中的指定行。下面的关系运算符都可以在 `if/goto` 语句中使用: `<`、`>`、`<=`、`>=`、`==` 和 `!=`。这些运算符的优先级和 C 语言中的优先级一样。

让我们看几个 Simple 程序。第一个 Simple 程序 (见图 15.21) 从键盘读取两个整数并存储在变量 `a` 和 `b` 中, 然后计算并打印出它们的和 (和存储在变量 `c` 中)。

命令	例句	描述
rem	50 rem this is a remark	rem 之后的文本都是说明文档，编译器会忽略这些内容
input	30 input x	显示一个提示用户输入一个整数的问号。读取从键盘输入的整数并存在变量 x 中
let	80 let u = 4 * (j-56)	把 4 * (j - 56) 的值赋给 u。注意任意复杂的表达式都可以在等号的右边
print	10 print w	显示 w 的值
goto	70 goto 45	把程序控制转移到第 45 行
if/goto	35 if i == z goto 80	比较 i 和 z 的值是否相等。如果相等就转到第 80 行，否则就继续执行后面的语句
end	99 end	程序结束

图 15.20 Simple 命令

```

1 10 rem determine and print the sum of two integers
2 15 rem
3 20 rem input the two integers
4 30 input a
5 40 rem
6 45 rem
7 50 rem add integers and store result in c
8 60 let c = a + b
9 65 rem
10 70 rem print the result
11 80 print c
12 90 rem terminate program execution
13 99 end

```

图 15.21 计算两个整数和的 Simple 程序

图 15.22 中的程序计算并打印出两个整数中的最大值。两个整数是从键盘输入的，分别存储在 s 和 t 中。if/goto 语句测试了条件  $s \geq t$ ，如果条件成立，转到第 90 行，输出 s；否则输出 t，并转到第 99 行，终止程序的执行。

```

1 10 rem determine the larger of two integers
2 15 input s
3 30 input t
4 32 rem
5 35 rem test if s >= t
6 40 if s >= t goto 90
7 45 rem
8 50 rem t is greater than s, so print t
9 60 print t
10 70 goto 99
11 75 rem
12 80 rem s is greater than or equal to t, so print s
13 90 print s
14 99 end

```

图 15.22 在两个整数中查找最大值的 Simple 程序

Simple 不提供循环语句（例如 C++ 中的 for、while 和 do/while）。但是，Simple 可以通过 if/goto 和 goto 语句来模拟循环语句。图 15.23 中的程序就用了一个标记控制的循环

来计算几个整数的平方。每一个从键盘输入的整数存储在  $j$  中, 如果它等于  $-9999$ , 就转到第 99 行终止程序的执行; 否则, 就把  $j$  的平方赋给  $k$ , 并将  $k$  输出到屏幕上, 然后把控制转移到第 20 行等待输入下一个数。

```

1 10  rem  calculate the squares of several integers
2 20  input j
3 23  rem
4 25  rem  test for sentinel value
5 30  if j == -9999 goto 99
6 33  rem
7 35  rem  calculate square of j and assign result to k
8 40  let k = j * j
9 50  print k
10 53 rem
11 55 rem  loop to get next j
12 60 goto 20
13 99 end

```

图 15.23 计算整数的平方

参考图 15.21、15.22 和 15.23 中的范例程序, 分别编写完成下列各个要求的 Simple 程序:

- a) 输入三个整数, 计算并打印出它们的平均值。
  - b) 用一个标记控制的循环输出 10 个数, 计算并打印出它们的和。
  - c) 用一个计数器控制的循环输入 7 个数 (输入值中既有正数也有负数), 计算并打印出它们的平均值。
  - d) 输入几个整数, 计算并打印出最大值。第一个数表示要处理的整数个数。
  - e) 输入 10 个数, 打印出最小值。
  - f) 计算并打印出从 2 到 30 的所有偶数的和。
  - g) 计算并打印出从 1 到 9 的所有奇数的乘积。
- 15.27 (建立一个编译器。要求先完成练习 5.18、5.19、15.12、15.13 和 15.26) Simple 语言已经在练习 15.26 中提出, 现在讨论一下如何建立 Simple 编译器。我们先考虑每个 Simple 程序转成 SML 后由 Simpletron 模拟器执行的过程 (见图 15.24)。编译器读取一个 Simple 程序文件, 把它转成 SML 代码, 这些代码输出到磁盘文件中 (每行都含有 SML 指令), 然后这个 SML 文件又被加载到 Simpletron 模拟器上, 输出的结果存入一个文件并显示在屏幕上。注意 Simpletron 程序是在练习 5.19 中开发的, 练习 5.19 中的程序要从键盘输入, 为了能够运行由编译器生成的文件, 必须把它改成可以从文件中读取数据。

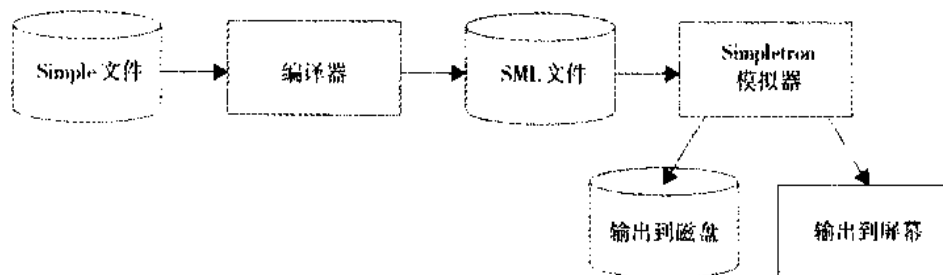


图 15.24 编写、编译和执行 Simple 程序

编译器分两步把一个 Simple 程序转换成 SML。第一步是构造一个符号表（对象），符号表把 Simple 程序中的每一个行号（对象）、变量名（对象）和常量（对象）与其在最终 SML 代码中的类型和相应的内存单元存储在一起（后面将详细讨论符号表）。第一步还为每条 Simple 语句（对象）产生了相应的 SML 指令对象。我们将会看到，如果 Simple 程序中包含把控制转移到其后行号的语句，则第一步编译产生的 SML 程序包含一些未完成的指令。第二步编译定位并完成这些指令，并把 SML 程序输出到文件中。

### 第一步编译

编译器首先把一条 Simple 语句读入内存。为了处理和编译，代码行必须被打断成单个的标记（也就是语句的各个部分，标准库函数 `strtok` 可用来完成这个任务），然后再重新处理这条语句。当编译器把一条语句打断成几个标记时，如果标记是行号、变量或常量，就把它放入符号表中。只有当行号是一个语句的第一个标记时，才把它放在符号表中。符号表是 `tableEntry` 对象的数组 `symbolTable`，它描述了程序中的每一个符号。程序中出现的符号的数量是没有限制的，因此特定程序的 `symbolTable` 可以很大。现在我们把 `symbolTable` 设成 100 个元素的数组（可以改变它的大小）。

每个 `tableEntry` 对象包含三个成员。`symbol` 是一个整数，该整数或者存放表示变量的 ASCII 码（这些变量名都是单个字符）、或者存放行号、或者存放常量。成员 `type` 是用下面的字符来表示的符号类型：'C' 表示常量、'L' 表示行号、'V' 表示变量。成员 `location` 存放了符号所要引用的 Simpletron 内存单元（00 到 99）。Simpletron 内存是一个包含 100 个整数元素的数组，数组中存放了 SML 指令和数据。对于行号，成员 `location` 是 Simpletron 内存数组中的 SML 指令（对应于该行号中的 Simple 语句）开始执行的内存单元。对于变量或常量，成员 `location` 是 Simpletron 内存数组中包含该变量或常量的内存单元。变量和常量从内存的后端开始分配，第一个存储在内存单元 99 中，第二个存储在内存单元 98 中等等。

符号表是 Simple 程序转换成 SML 的桥梁。我们在第 7 章中已经了解到，SML 指令是四位整数，由操作码和操作数两部分组成。操作码由 Simple 的命令决定，例如：命令 `input` 对应于 SML 操作码 10（读），`print` 对应于 SML 操作码 11（写）。操作数是一个内存单元，存储着操作码所需的数据（如操作码 10 从键盘读入一个值，存储到操作数指定的内存单元中）。编译器通过查找符号表（`symbolTable`）找出每个符号相应的内存单元，从而执行 SML 指令。

对每条 Simple 语句的编译取决于其命令。例如，把一条 `rem` 语句的行号存入符号表后，对于余下的部分，编译器就忽略了，因为注释仅仅是说明性文档。`input`、`print`、`goto` 和 `end` 语句对应于 SML 中的 `read`、`write`、`branch`（转移到一个特定的内存单元）和 `halt` 指令。包含这些 Simple 指令的语句都直接转换成相应的 SML 指令（注意，`goto` 语句如果要转移到的其后的语句，有可能包含一个暂时不能解决的引用，有时把这种引用称为提前引用）。

当编译一个带有引用未解决的 `goto` 语句时，为指示第二步编译必须完成这条指令，SML 指令必须做个标志。这些标志存储在包含 100 个元素的整数数组 `flags` 中（每个元素初始化值都是 -1）。如果 Simple 程序的行号所引用的内存单元还是未知数的时候（例如它还没有存入符号表中），这个行号就被存储到数组 `flags` 的元素中，该元素的下标与未



完成的指令所在的下标相同。未完成指令中的操作数暂时置成00。例如,一个无条件转移指令(提前引用)在第二步编译之前让它保持+4000。

if/goto和let语句的编译要比其他语句的编译复杂得多,它们是仅有的几个产生不止一条SML指令的语句。对于一条if/goto语句,编译器将产生测试其条件的代码和必要的分支代码,分支代码将产生一个未解决的引用。每一个关系运算符和相等运算符都可以用SML转移到零分支和转移到负分支的指令(也可能两者都用)来模拟。

对于let语句,编译器产生的代码可计算任意复杂程度的包含整数变量或常量的算术表达式。表达式中的每个操作数和运算符应该用空格分开。练习15.12和15.13描述了中缀到后缀的转换算法以及编译器用后缀算法来计算表达式。在建立编译器之前,应该先完成这些练习。编译器在遇到一个表达式时,会把它从中缀形式转换成后缀形式,然后再计算它的值。

编译器是怎样产生计算包含变量的表达式的机器语言?后缀计算算法是一个“异常指令”(hook),它允许编译器产生SML指令,而不是真的去计算表达式。为了使编译器里的hook起作用,后缀计算算法必须修改,使它能够对它遇到的每个符号都去查找符号表(也可能要把符号添加到符号表中)并找出这个符号对应的内存单元,然后把内存单元(而不是符号)压入栈顶。当遇到后缀表达式中的某个运算符时,就把栈顶的两个内存单元弹出,产生以这两个内存单元为操作数的机器语言。每一个子表达式的结果先存储在临时内存单元里,然后再把它压回堆栈,使得后缀表达式的计算能够继续进行。当后缀表达式计算结束,堆栈里就只留下一个包含结果的内存单元,把该结果弹出,产生把它赋给let语句左边变量的SML指令。

## 第二步编译

第二步编译要完成两个任务,一是解决那些未解决的引用,二是把SML代码输出到文件中。按如下步骤解决引用问题:

- a) 从数组flags中找出一个未解决的引用(也就是值不等于-1的元素)。
- b) 从数组simpleTable中找到包含保存在数组flags中的符号的对象(确定行号的符号类型是'L')。
- c) 把结构成员location的内存单元插入到未解决引用的指令中(别忘了每个含有未解决引用的指令中都有操作数00)。
- d) 重复上述1、2、3步,直到flags中的元素都被处理完。

处理完未解决的引用后,包含SML代码的整个数组被输出到一个磁盘文件中,文件中每一行都含有一条SML指令。Simpletron模拟器可以读取这个文件然后运行它(当然这个模拟器已经修改成可以从一个文件中读取数据)。

## 一个完整的范例

下面的范例反映了Simple编译器把Simple程序完全转换成SML程序的过程。考察一个Simple程序,该程序要求输入一个整数值,然后计算从1到该整数值的累加和。图15.25列出了该程序以及第一步编译产生的SML指令。图15.26列出了第一步编译所构造的符号表。

Simple 程序	SML 内存单元和指令	描述
5 rem sum 1 to x	无	忽略 rem
10 input x	00 +1099	把 x 读入内存单元 99
15 rem check y == x	无	忽略 rem
20 if y == x goto 60	01 +2098	把 y (98) 装入累加器
	02 +3199	从累加器中减去 x (99)
	03 +4200	零分支表示未解决的内存定位
25 rem increment y	无	忽略 rem
30 let y = y + 1	04 +2098	把 y 装入累加器
	05 +3097	把累加器加 1 (97)
	06 +2196	存储到临时内存单元 96
	07 +2096	装入临时内存单元 96 的内容
	08 +2198	把累加器中的内容存入 y
35 rem add y to total	无	忽略 rem
40 let t = t + y	09 +2095	把 t (95) 装入累加器
	10 +3098	把 y 与累加器的内容相加
	11 +2194	存储在临时内存单元 94
	12 +2094	装入临时内存单元 94 的内容
	13 +2195	把累加器内容存入 t
45 rem loop y	无	忽略 rem
50 goto 20	14 +4001	转移到内存单元 01
55 rem output result	无	忽略 rem
60 print t	15 +1195	把 t 输出到屏幕
99 end	16 +4300	终止执行

图 15.25 第一步编译后生成的 SML 指令

符号	类型	内存单元
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

图 15.26 图 15.25 中程序的符号表

大部分 Simple 语句都被直接转换成了 SML 指令, 注释语句、第 20 行的 if/goto 语句以及 let 语句则是例外。注释不需要被转成机器语言, 但是注释语句的行号要放入到符

号表中,以防止 goto 或 if/goto 语句引用,第 20 行的代码指出,如果  $y == x$  为 true,程序控制就转移到第 60 行,因为第 60 行在其后面,所以第一步编译还没有把 60 放到符号表中(行号只在作为语句的头一个标记时才被放入符号表中)。因此,现在还不可能找到 SML 指令数组中 03 单元内 SML 零分支指令的操作数,为指示第二步编译来完成这条指令,编译器把 60 放在 flags 数组中的 03 单元处。

因为 Simple 语句和 SML 指令不是一一对应的,所以我们必须跟踪下一条指令在 SML 数组中的内存单元。例如,第 20 行的 if/goto 语句被编译成 3 条 SML 指令。每当产生一条 SML 指令,我们必须把指令计数器加 1 以指示下一条指令在 SML 数组中的内存单元。注意,对于一个带有很多语句、变量和常量的 Simple 程序,Simpletron 内存的容量也许会成为一个问题。编译器内存溢出是可能的,为了检测这种情况,程序应该包含一个数据计数器来记录要存放在 SML 数组中的下一个变量或常量的内存单元。如果指令计数器的值大于数据计数器的值,说明 SML 数组满了。编译器这时就应该终止编译,并打印出指示编译时内存溢出的出错消息。这是为了强调程序员虽然可以通过编译器解除分配内存的任务,但程序员必须小心地判断内存中的指令位置和数据,而且必须检查编译操作时内存溢出等错误。

### 一步一步观看编译过程

让我们完整地看一看图 15.25 中的程序的编译过程。编译器把程序第一行读入内存:

```
5 rem sum 1 to x
```

该语句的第一个标记(行号)可以用函数 strtok 找出(见第 5 章和第 16 章对 C++ 字符串操作函数的讨论)。函数 strtok 返回的标记可以用函数 atoi 转换成一个整数,因此可以在符号表中定位符号 5。如果在符号表中找不到这个符号,就把它插进去。因为这是程序的第一行,所以符号表中还没有这个符号。因而 5 要作为类型 'L' (行号) 插入到符号表中,并且存到 SML 数组的第一个内存单元(00)。尽管这是一个注释行,符号表还是为它的行号分配一个空间(也许 goto 或 if/goto 语句要引用它)。rem 语句并不产生 SML 指令,因此指令计数器不加 1。

然后把语句:

```
10 input x
```

打断成标记。行号 10 被放入到符号表中,类型是 L,并给它分配 SML 数组中的第一个内存单元(因为程序最开始是一条注释,所以指令计数器的当前值是 00)。input 命令意味着下一个标记是一个变量(只有变量才能出现在 input 语句中)。因为 input 可以直接对应于一个 SML 操作码,所以编译器仅仅需要确定 x 在 SML 数组中的内存单元。因为在符号表中找不到 x,所以把 x 的 ASCII 码值插入到符号表中,类型是 V,给它分配 SML 数组中的内存单元 99(数据按从后向前的顺序从内存单元 99 开始存储)。现在可以为这条语句产生 SML 代码。构成完整指令的方法是:操作码 10 (SML 中的读操作码)乘以 100,再加上在符号表中确定的 x 的内存单元。这条指令存在 SML 数组中的内存单元 00。因为产生了一条 SML 指令,所以把指令计数器加 1。

然后再把语句:

```
15 rem check y == x
```

打断成标记。在符号表中查找行号 15，因为没找到，所以把行号输入到符号表中，类型是 L，并分配到数组的下一个内存单元 01 中（rem 语句不产生 SML 代码，所以指令计数器不加 1）。

然后又把语句：

```
20 if y == x goto 60
```

打断成标记。行号 20 被插入到符号表中，类型是 L，被分配到数组中 01 单元的下一个单元。if 命令表示要计算一个条件。在符号表中找到不到变量 y，因此它被插入到符号表中，类型是 V，分配到内存单元 98。接下来生成计算这个条件的 SML 指令。因为 SML 中没有与 if/goto 语句等价的指令，所以必须通过计算 x 和 y，根据结果的分支来模拟这条语句。如果 y 等于 x，则 y 减 x 等于 0，因此零分支指令可用来模拟 if/goto 语句。第一步先把 y 装入累加器，产生指令 01 +2098；第二步，从累加器中减去 x，产生指令 02 +3199，累加器的值可能是 0、正数或是负数。因为运算符是 ==，所以我们是希望是零分支。首先，在符号表中查找分支内存单元（此处是 60），因为没有找到，所以将 60 存放到 flags 数组中的位置 03，并产生指令 03 +4200（我们不能加入分支单元，因为还没有在 SML 数组中为 60 分配一个单元），指令计数器加到了 04。

编译器继续处理语句：

```
25 rem increment y
```

行号 25 插入到符号表中，类型是 L，并分配内存单元 04，指令计数器不增加。

当把语句：

```
30 let y = y + 1
```

打断成标记时，行号 30 被插入到符号表，类型是 L，给它分配 SML 内存单元 04。命令 let 表示该行是一条赋值语句。首先把该行的所有的符号插入到符号表中（如果符号表中没有）。整数 1 加到符号表中，类型是 C，分配内存单元 97。接下来把赋值语句的右边从中缀形式转换成后缀形式，然后计算后缀表达式（y 1 +）。定位符号 y 在符号表中的内存单元并把内存单元压入堆栈，再定位符号 1 在符号表中的内存单元并把相应内存单元也压入堆栈。在碰到运算符 + 时，就从堆栈中分别弹出该运算符的右操作数和左操作数，然后产生 SML 指令：

```
04 +2098 (装入 y)
05 +3097 (加 1)
```

结果存入到临时内存单元（96）中并压入堆栈，指令如下：

```
06 +2196 (临时存储)
```

并将临时地址压入堆栈中。现在表达式已计算完毕，所以必须把结果存入 y（即运算符 = 左边的变量）。因此把临时单元装入累加器，再把累加器的内容存入 y，指令如下：

```
07 +2096 (装入暂存)
08 +2198 (存入 y)
```

读者立即会注意到，SML 指令出现冗余，我们很快就会讨论这个问题。

当把语句:

```
35 rem add y to total
```

打断成标记时,行号 35 插入到符号表中,类型是 L,分配内存单元 09。

语句:

```
40 let t = t + y
```

很像第 30 行语句。把变量 t 插入到符号表中,类型是 V,分配内存单元 95。接下来的指令在逻辑上和格式上都和第 30 行语句相似,产生指令 09+2095、10+3098、11+2194、12+2094 和 13+2195。注意, t+y 的结果在赋给 t (95) 之前,先被暂存到内存单元 94 中。读者又一次发现内存单元 11、12 中的指令出现冗余,我们一会儿就会讲到这个问题。

语句:

```
45 rem loop y
```

是一条注释,因此行号 45 被加到符号表中,类型是 L,分配内存单元 14。

语句:

```
50 goto 20
```

把控制转移到第 20 行。把行号 50 插入到符号表中,类型是 L,分配内存单元 14。goto 语句的等价语句是无条件转移 (40) 指令 (把控制转移到指定的位置)。编译器在符号表中找到第 20 行的内存单元 01,把操作码 (40) 乘以 100 再加上单元 01 就生成了指令 14+4001

语句:

```
55 rem output result
```

也是一条注释,因此行号 55 被加到符号表中,类型是 L,分配的内存单元是 15。

语句:

```
60 print t
```

是一条输出语句。行号 60 被插入到符号表中,类型是 L,分配内存单元 15。操作码 11 (写) 与 print 等价。t 的位置再加上操作码乘以 100 就生成了指令。

语句:

```
99 end
```

是程序的最后一行。行号 99 被插入到符号表中,类型是 L,分配的内存单元是 16。命令 end 产生的 SML 指令是 +4300 (43 是 SML 中的停机码),该指令是作为最后一条指令写入 SML 数组中。

这样就完成了编译器的第一步编译。现在观察第二步编译。查找数组 flags 中非 -1 的项。因为内存单元 03 中存储了 60,所以编译器知道指令 03 还没有完成。编译器从符号表中找到 60,确定其内存单元并把它加到未完成的指令中以完成这条指令。在本例中,行号 60 的内存单元是 15,因此用完整的指令 03+4215 来代替原来的 03+4200。现在,这个程序就成功地编译了。

为了建立一个编译器,必须做以下工作:

- a) 修改在练习 5.19 中编写的 Simpletron 模拟程序, 让它可以从用户指定的文件中输入数据 (见第 14 章), 并且还要能够像在屏幕上显示的格式那样把结果输出到文件中。将模拟程序变成面向对象程序。特别是将每部分硬件作为一个对象。将指令类型用继承变成类层次。然后多态执行程序, 用 executeInstruction 消息让每条指令执行。
- b) 修改练习 15.12 的中缀转换成后缀的计算算法, 以便处理多位整数操作数和单字母变量名操作数。提示: 标准库函数 strtok 可用来定位表达式中的每一个常量和变量, 标准库函数 atoi 可把一个字符串常量转换成整数 (注意: 后缀表达式的数据表示必须要改成支持变量名和整数常量)。
- c) 修改后缀计算算法以处理多位整数操作数和变量名操作数。为了能够不直接计算表达式就生成 SML 指令, 算法还应该实现前面讨论的 “hook”。提示: 标准库函数 strtok 可用来定位表达式中的常量和变量, 标准库函数 atoi 可把一个字符串常量转换成整数 (注意: 后缀表达式的数据表示必须要改成支持变量名和整数常量)。
- d) 建立编译器, 把 b)、c) 部分用于 let 语句中表达式的计算。程序中应该包括一个执行第一步编译的函数和第二步编译的函数。这两个函数可调用其他函数来完成其任务。
- 15.28 (优化 Simple 编译器) 程序经过编译转换成 SML 程序后, 就会产生一组指令。指令的某种组合经常会多次重复出现 (通常是三个一组), 这样一组指令称为 “结果生成指令” (production)。一组结果生成指令通常是由如装入、相加和存储这样的三条指令组成的。例如, 图 15.27 描述了三条 SML 指令, 它们是在编译图 15.25 中的程序时产生的。前三条指令是一组结果生成指令, 它把 1 加到 y 上。注意, 指令 06 和 07 先把累加器值存储到临时单元 96 中, 再把该值装回到累加器, 这样指令 08 就能把这个值存入内存单元 98 中。一组结果生成指令后面经常跟有一条装入指令以操作刚才存储的单元。取消存储指令和其后对同一个内存单元操作的装入指令可优化这段代码。因为程序中的指令比原来少了, 所以这种优化可以使 Simpletron 更快地运行程序。图 15.28 是对图 15.25 中的程序优化后的 SML 代码, 可以看到, 优化代码中少了四条指令, 内存空间节省了 25%。
- 修改编译器, 增加一个优化 Simpletron 机器语言代码的选项。比较非优化代码和优化代码, 计算代码减少的百分数。

```

14 04 +2098 (装入)
15 05 +3097 (相加)
16 06 +2197 (存储)
17 07 +2096 (装入)
18 08 +2198 (存储)

```

图 15.27 图 15.25 中程序中非优化代码

Simple 程序	SML 内存单元和指令	描述
5 rem sum 1 to x	无	忽略 rem
10 input x	00 +1099	把 x 读到内存单元 99
15 rem check y == x	无	忽略 rem
20 if y == x goto 60	01 +2098	把 y (98) 装入累加器
	02 +3199	从累加器中减去 x (99)
	03 +4211	如果为 0 转移到内存单元 11
25 rem increment y	无	忽略 rem
30 let y = y + 1	04 +2098	把 y 装入累加器
	05 +3097	把累加器加 1 (97)

(续表)

Simple 程序	SML 内存单元和指令		描述
	06	+2198	把累加器值存入 y (98)
35 rem add y to total	无		忽略 rem
40 let t = t + y	07	+2096	把内存单元 (96) 装入 t
	08	+3098	给累加器加 y (98)
	09	+2196	把累加器值存储入 t (96)
45 rem loop y	无		忽略 rem
50 goto 20	10	+4001	转移到内存单元 01
55 rem output result	无		忽略 rem
60 print t	11	+1196	把 t (96) 输出到屏幕
99 end	12	+4300	终止执行

图 15.28 图 15.25 中的程序优化后的代码

15.29 (修改 Simple 编译器) 对 Simple 编译器做如下的修改。有些修改可能还要对练习 5.19 中的 Simpletron 模拟器做相应的改动。

- 使 let 语句中能够使用求模运算符 (%), 对应的 Simpletron 机器语言中必须加入求模指令。
- 使 let 语句中能够使用幂运算符 (^), 对应的 Simpletron 机器语言中必须加入幂运算指令。
- 使编译器能够同时辨认 Simple 语句中的大小写字母 (即 'A' 和 'a' 是等价的), 无需修改 Simpletron 模拟器。
- 使 input 语句能够为多个变量读取值 (如 input x, y), 无需修改 Simpletron 模拟器。
- 使编译器能在一条 print 语句中输出多个值 (如 print a, b, c), 无需修改 Simpletron 模拟器。
- 给编译器添加语法检查功能, 当在 Simple 程序中遇到语法错误时输出报错消息, 无需修改 Simpletron 模拟器。
- 允许使用整型数组, 无需修改 Simpletron 模拟器。
- 允许使用子程序, 子程序由 Simple 命令 gosub 和 return 指定。命令 gosub 将控制交给子程序, 命令 return 把控制交给 gosub 后面的语句, 这类似于 C++ 中的函数调用, 同一个子程序可被程序中的多处 gosub 调用。无需修改 Simpletron 模拟器。
- 允许使用如下形式的循环结构:

```
for x = 2 to 10 step 2
    Simple 语句
next
```

这条 for 语句从 2 循环到 10, 步长为 2。next 行表示 for 循环的结束。无需修改 Simpletron 模拟器。

- 允许使用如下形式的循环结构:

```
for x = 2 to 10
    Simple 语句
next
```

这条 for 语句从 2 循环到 10, 默认步长为 1。无需修改 Simpletron 模拟器。

- 允许编译器处理字符串的输入和输出。这需要对 Simpletron 模拟器加以修改使它能够处理和存储字符串。提示: 每个 Simpletron 字可被分成两组, 每组包含一个两位数字

的整数, 每个两位数字整数表示与字符 ASCII 码等值的十进制数。添加一条机器语言指令, 使它打印出从 Simpletron 某个内存单元开始的一个字符串, 起始单元中的前半个字是字符串中的字符个数 (即字符串的长度), 其后的每半个字都是用两位十进制数字表示的 ASCII 字符。这条机器语言指令检查字符串的长度, 然后把每个两位数字转化成等价的字符并打印出来。

l) 除了整数外, 允许编译器还能处理浮点数。必须对 Simpletron 模拟器作相应的修改使它能够处理浮点数。

15.30 (Simple 解释器) 解释器是一个程序, 它读入一条用高级语言编写的语句, 确定这条语句所要执行的操作后立即执行这个操作, 而不是先把整个程序转换成机器语言。因为程序中遇到的每条语句都要先予以解释, 所以用解释器运行程序比较慢。如果语句出现在循环中, 那么每执行一次循环时都要对这些语句重复解释。早期的 BASIC 语言就是用解释器执行的。

为练习 15.26 中讨论的 Simple 语句编写一个解释器。程序应该用练习 15.12 中开发的中缀-后缀转换算法和练习 15.13 中开发的后缀计算算法计算 let 语句中的表达式, 练习 15.26 中对 Simple 语言的限制同样适用于这个程序。用在练习 15.26 中编写的 Simple 程序测试解释器, 比较程序在解释器中的执行结果和在 Simpletron 模拟器 (见练习 5.19) 中编译执行的结果。

15.31 (在链表中任何地方插入/删除) 我们的链表类模板只能在链表前面和后面插入/删除。这些功能在使用私有继承和复合产生堆栈类模板和队列类模板时很方便, 可以使代码最小化, 只要复用链表类模板即可。实际上, 链表是更一般性的。修改本章的链表类模板, 以便在链表中任何地方插入/删除。

15.32 (不带尾指针的链表与队列) 我们的链表 (图 15.3) 使用 firstPtr 和 lastPtr。lastPtr 用于 List 类的 insertAtBack 和 removeFromBack 成员函数。insertAtBack 函数对应于 Queue 类的 enqueue 成员函数。改写 List 类, 使它不用 lastPtr。这样, 链表末尾的任何操作都要从表头开始查找。这样会不会影响 Queue 类的实现方法 (图 15.12)?

15.33 用复合版本的堆栈程序 (图 15.11) 建立完整运行的堆栈程序。将这个程序修改成内联成员函数。比较两种方法。总结内联成员函数的优点与缺点。

15.34 (进行二叉树排序与查找) 进行二叉树排序的一个问题是数据插入的顺序会影响树的形状, 对于相同的数据集合, 不同排序可能得到形状大不相同的二叉树。二叉树排序与查找的性能和二叉树的形状关系很大。如果数据以升序或降序插入, 二叉树形状如何? 什么样的二叉树形状能达到最大的查找性能?

15.35 (索引表) 文中曾介绍过, 链表应顺序查找。对于大型链表, 这样查找的性能很差。一个改进链表查找性能的常用方法是生成和维护索引表。索引是表中各个关键字位置的指针集合。例如, 查找大型姓名链表的应用程序可以通过建立 26 个项目 (各对应一个字母) 的索引来提高性能。姓氏以 Y 开头的项目查找首先通过索引找到 "Y" 项目开头的位置, 然后跳到表中这个位置, 开始线性查找, 直到找出所要项目。这样比从头开始查找会快得多。利用图 15.3 的 List 类为基础生成 IndexedList 类。编写一个程序, 演示索引表的操作。一定要包括成员函数 insertInIndexedList、searchIndexedList 和 deleteFromIndexedList。



## 第16章 位、字符、字符串和结构

### 教学目标

- 能够建立和使用结构
- 能够按值和按引用调用向函数传递结构
- 能够用位运算符操作数据并能够建立紧凑存储数据的位段
- 能使用字符处理库 (ctype.h) 的函数
- 能使用一般实用程序库 (stdlib.h) 的字符串转换函数
- 能使用字符串处理库 (string.h) 的字符串处理函数
- 体会函数库在实现软件复用中的功能

### 16.1 简介

本章要更多地介绍结构, 然后讨论位、字符和字符串的操作。这里介绍的许多方法都是类C语言的、用于C++程序员处理C语言遗留代码。

结构中可包含多种不同数据类型的变量(而数组中只包含相同数据类型的元素)。这里对结构的介绍也适用于类。C++中结构与类的惟一差别在于结构成员默认为公有访问, 而类成员默认为私有访问。结构通常用来定义存储在文件中的记录(见第14章)。指针和结构可用来构造更复杂的数据结构, 如链表、队列、堆栈和树(见第15章)。我们要介绍如何声明结构、初始化结构以及将结构传递给函数, 然后介绍一个高效的“洗牌和发牌”模拟程序。

### 16.2 结构的定义

考虑如下的结构定义:

```
struct Card {  
    char *face;  
    char *suit;  
};
```

关键字 struct 引出了结构 Card 的定义。标识符 Card 是结构“名字”(structure name), 命名了结构的定义, 它与关键字 struct 一起使用可用来声明结构类型的变量。就本例而言, 结构的类型是 struct Card。在定义结构体的花括号内声明的变量是结构的成员。同一个结构的成员不能有相同的名字, 但是两个不同的结构可包含同名的成员而不会冲突。每一个结构定义必须用分号结束。

#### 常见编程错误 16.1

忘记终止结构定义的分号。

Card的定义包含两个类型为char\*的成员face和suit。结构的成员可以是int、float等等的基本数据类型的变量,也可以是数组和其他结构等等的聚合体。数组的每一个元素必须是同一种数据类型(见第4章),而结构的成员可以是各种数据类型。例如,结构Employee可能会包括:表示姓、名的字符串成员、表示雇员年龄的一个int类型的成员、表示性别的char类型成员('M'或'F')以及表示雇员工资的float类型的成员等等。

结构中不能包含本身的实例。例如,不能在结构Card的定义中声明结构变量Card,但是可以包含指向结构Card的指针。如果一个结构包含指向同一个结构类型的指针成员,那么这种结构称为“自引用结构”(self-referential structure)。第15章用自引用结构建立了各种链式数据结构。

编译器不为上述结构定义保留任何内存空间,而是建立了用于声明变量的一种新的数据类型。声明结构变量与声明其他类型变量类似。声明语句:

```
Card a, deck[ 52 ], *c;
```

声明a为Card类型的变量、把deck声明为有52个元素的Card类型的数组、c是指向结构Card的指针。也可以用如下方法声明给定结构类型的变量:在结构定义的花括号之后用逗号分开的变量名列表,并在最后加上结束结构定义的分号。例如,可像下面那样在定义结构时声明上述变量:

```
struct Card {  
    char *face;  
    char *suit;  
} a, deck[ 52 ], *c;
```

也可以在定义结构时不使用结构名。如果结构定义中没有包含结构名,那么该结构类型的变量只能在定义结构时声明,而不能单独声明。

#### 编程技巧 16.1

在建立结构类型时提供结构名。以后可以方便地用结构名声明该结构类型的新变量,而且在将结构作为参数传递给函数时也会用到。

对结构的正确的操作是:把结构变量赋给同一种类型的结构变量、获取结构变量的地址、访问结构变量的成员(见第6章)以及用sizeof运算符确定结构变量的大小。和类一样,大多数运算符可以重载,用于结构类型的对象。

因为结构成员不一定是连续存放在内存中的字节,所以不能比较结构。因为计算机可能会把指定类型的数据只沿某种存储边界(boundary)存放,如半字、一个字或双字长边界(字是用来存储数据的标准内存单元,通常占2个字节或4个字节),所以有时在结构中存在“空洞”。例如,如下的结构定义声明了Example类型的变量sample1和sample2:

```
struct Example {  
    char c;  
    int i;  
} sample1, sample2;
```

字长为两个字节的计算机可能要求把Example的每一个成员从字的边界开始存放,即存放在字的开始位置(这个特点与机器有关)。图16.1是Example类型的变量(已将字符'a'和整数97赋给该变量)在内存中的存放情况,图中显示出了存放在内存中的代表这两个值的位。如果结构的成员从字的边界开始存放,那么就会有一个字节的“空洞”(图中的字节1),“空洞”中的值是不确定的,如

果 sample1 和 sample2 的成员值确实是相等的,但是比较的结构却不一定是相等的,因为未定义的一个字节的“空洞”中不一定就包含相等的值。

#### 常见编程错误 16.2

因为不同的系统对结构存放的方式不同,所以比较结构是一种语法错误。

#### 可移植性提示 16.1

特定类型数据项的大小以及存储方式都与机器有关,结构就是这样一种情况。

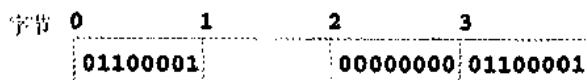


图 16.1 Example 类型的变量可能的存储方式,图中显示了一个没有确定值的存储区域

## 16.3 结构的初始化

和数组一样,结构也可以用初始化值列表初始化,即在声明结构的变量时,在变量名后用等号连接括在花括号中的初始化值列表,初始化值用逗号分开。例如,声明语句:

```
Card a = { "Three", "Hearts" };
```

建立了类型为 Card(前面已定义好)的变量 a,并把成员 face 和 suit 分别初始化为 "Three" 和 "Hearts"。如果初始化值的个数少于结构中的成员数目,剩余的成员自动初始化为 0。在函数定义之外声明的结构变量(即外部变量)如果没有在外部声明中显式地初始化,就初始化为 0。也可以用赋值语句初始化结构变量,即赋给它同一种类型的结构变量,或指定结构的单个成员的值。

## 16.4 函数和结构

把结构传递给函数的方式有两种:传递单个成员、传递整个结构。默认情况下,把整个结构或单个成员(除了单个数组成员)传递给函数是按值调用。也可以按引用调用传递结构或成员,传递指向结构的指针或引用。

传递结构变量的地址可实现结构的按引用调用。和其他数组一样,结构数组也是自动按引用方式调用的。

第4章曾介绍过能够用结构实现数组的按值调用传递。建立一个把数组作为成员的结构(或类)即可实现按值调用传递数组。这是因为结构是按值调用传递的,所以传递的数组也是按值调用。

#### 常见编程错误 16.3

以为传递结构和传递数组一样是自动实现按引用调用的,因而试图在被调用函数中修改调用函数中的结构。

#### 性能提示 16.1

以按引用调用方式传递结构比按值调用方式传递结构效率更高。以按值调用方式传递结构的需要对整个结构做一份副本。

## 16.5 类型定义: typedef

关键字 typedef 可用来建立已定义好的数据类型的别名。为建立较短的类型名, 程序员经常用 typedef 定义结构名。例如, 下列语句:

```
typedef Card *CardPtr;
```

定义新的类型名 CardPtr, 它是类型 Card\* 的别名。

### 编程技巧 16.2

为了强调用 typedef 定义的类型名是其他类型名的别名, 以大写形式书写用 typedef 定义的类型名。

用 typedef 建立一个新的名字实际上并没有建立一个新的类型, 而只是建立了一个用作现有类型名别名的新类型名。

经常用 typedef 建立内部数据类型的别名。例如, 需要 4 字节整数的程序在一种系统上可能要用类型 int, 而在另一个系统上 (2 字节整数) 可能要用类型 long int。为可移植性而设计的程序经常用 typedef 建立 4 字节整数的别名 (如 Integer)。Integer 可以作为 4 字节整数系统中 int 的别名, 也可以作为 2 字节整数系统中 long int (long int 占 4 字节) 的别名。为了编写可移植的程序, 程序员只需要声明所有的 4 字节整数变量为 Integer 类型。

### 可移植性提示 16.2

使用 typedef 可使程序具有更好的可移植性。

## 16.6 范例: 高效的洗牌和发牌模拟程序

图 16.2 中的程序建立在第 5 章讨论的发牌和洗牌模拟程序的基础上。程序用结构数组表示一副牌, 并使用了高效的发牌和洗牌模拟算法。程序输出见图 16.3。

```
1 // Fig. 16.2: fig16_02.cpp
2 // Card shuffling and dealing program using structures
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6 #include <time.h>
7
8 struct Card {
9     char *face;
10    char *suit;
11 };
12
13 void fillDeck( Card *, char *[], char *[] );
14 void shuffle( Card * );
15 void deal( Card * );
16
17 int main()
18 {
19     Card deck[ 52 ];
20     char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
```

```

21         "Six", "Seven", "Eight", "Nine", "Ten",
22         "Jack", "Queen", "King" };
23     char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
24
25     srand( time( 0 ) );           // randomize
26     fillDeck( deck, face, suit );
27     shuffle( deck );
28     deal( deck );
29     return 0;
30 }
31
32 void fillDeck( Card *wDeck, char *wFace[], char *wSuit[] )
33 {
34     for ( int i = 0; i < 52; i++ ) {
35         wDeck[ i ].face = wFace[ i % 13 ];
36         wDeck[ i ].suit = wSuit[ i / 13 ];
37     }
38 }
39
40 void shuffle( Card *wDeck )
41 {
42     for ( int i = 0; i < 52; i++ ) {
43         int j = rand() % 52;
44         Card temp = wDeck[ i ];
45         wDeck[ i ] = wDeck[ j ];
46         wDeck[ j ] = temp;
47     }
48 }
49
50 void deal( Card *wDeck )
51 {
52     for ( int i = 0; i < 52; i++ )
53         cout << setiosflags( ios::right )
54              << setw( 5 ) << wDeck[ i ].face << " of "
55              << setiosflags( ios::left )
56              << setw( 8 ) << wDeck[ i ].suit
57              << ( ( i + 1 ) % 2 ? '\t' : '\n' );
58 }

```

图 16.2 高效的洗牌和发牌模拟程序

函数fillDeck按每种花色从“A”到“K”的顺序初始化数组Card。数组Card传递给函数shuffle（函数shuffle完成了高性能的发牌算法）。函数shuffle取52个Card结构的数组作为参数，并用for循环结构循环通过52张牌（数组下标0~51）。程序为每一张牌选择一个0~51之间的随机数，然后把数组中当前的Card结构与随机选出的Card结构交换。循环通过整个数组共交换牌52次，从而“洗”好牌（Card结构的数组），这种算法不会像第5章的洗牌算法那样发生无限延迟现象。因为Card结构被交换到数组中合适的位置上，所以在函数deal中实现的高效发牌算法只需要一次就能够发出洗好的牌。

#### 常见编程错误 16.4

在引用结构数组中的单个结构时忘记使用数组下标。

Eight of Diamonds	Ace of Hearts
Eight of Clubs	Five of Spades
Seven of Hearts	Deuce of Diamonds
Ace of Clubs	Ten of Diamonds
Deuce of Spades	Six of Diamonds
Seven of Spades	Deuce of Clubs
Jack of Clubs	Ten of Spades
King of Hearts	Jack of Diamonds
Three of Hearts	Three of Diamonds
Three of Clubs	Nine of Clubs
Ten of Hearts	Deuce of Hearts
Ten of Clubs	Seven of Diamonds
Six of Clubs	Queen of Spades
Six of Hearts	Three of Spades
Nine of Diamonds	Ace of Diamonds
Jack of Spades	Five of Clubs
King of Diamonds	Seven of Clubs
Nine of Spades	Four of Hearts
Six of Spades	Eight of Spades
Queen of Diamonds	Five of Diamonds
Ace of Spades	Nine of Hearts
King of Clubs	Five of Hearts
King of Spades	Four of Diamonds
Queen of Hearts	Eight of Hearts
Four of Spades	Jack of Hearts
Four of Clubs	Queen of Clubs

图 16.3 高效的洗牌和发牌模拟程序的输出

## 16.7 位运算符

C++ 提供了大量位操作功能，程序员可以进行位和字节操作。操作系统、测试设备软件、网络软件和许多其他软件要求程序员“直接与硬件通信”。本节和下面几节介绍 C++ 的位操作功能。我们将介绍 C++ 的许多位运算符和如何用位段节省内存。

所有的数据在计算机内部都是用位序列表示的，每一位的值为 0 或 1。在大多数系统中，连续的 8 位构成一个字节，它是一个 char 类型变量的标准存储单位，其他类型的数据要用更多的字节数存储。位运算符用来操作整数操作数的位，整数操作数的类型包括有符号 (signed) 和无符号 (unsigned) 的 char、short、int 和 long 类型。位运算符通常用于 unsigned 类型的整数。

### 可移植性提示 16.3

位运算与机器有关。

本节讨论的位运算符反映了整数操作数的二进制表示。二进制 (基数为 2) 数值系统的详细解释见附录“数值系统”。16.7 节和 16.8 节的程序是用 Borland C++ 在 PC 兼容机上测试的。这种系统采用 16 位 (2 个字节) 存储整数。由于位运算与机器有关，这些程序可能不能在读者的系统上运行。

位运算符包括按位与 (&)、按位或 (|)、按位异或 (^)、左移位 (<<)、右移位 (>>) 和按位取反 (~)。按位与、按位或和按位异或运算符按位比较两个操作数。如果两个操作数的相应位为 1，按位与运算符把结果中的对应位设置为 1。如果两个操作数中的一个或两个的相应位为 1，按位或

运算符把结果中的对应位设置为1。如果两个操作数的相应位只有一个为1，按位异或运算符把结果中的对应位设置为1。左移位运算符把其左操作数的位向左移动右操作数指定的位数。右移位运算符把其左操作数的位向右移动右操作数指定的位数。按位取反运算符把其操作数中所有值为0和所有值为1的位分别在结果的相应位中设置为1和0。下面几个例子详细地讨论了每一种位运算符的使用。图16.4归纳了这些位运算符的用法。

运算符	名称	描述
&	按位与	如果两个操作数的相应位都为1，结果中的相应位设置为1
	按位或	如果两个操作数的相应位有一个为1，结果中的相应位设置为1
^	按位异或	如果两个操作数的相应位只有一个为1，结果中的相应位设置为1
<<	左移位	把第一个操作数的位向左移动第二个操作数指定的位数，右边空出的位补0
>>	右移位	把第一个操作数的位向右移动第二个操作数指定的位数，填补左边空出位的方法与机器有关
~	按位取反	所有0位设置为1，所有1位设置为0

图 16.4 位运算符

在使用位运算符时，为准确地说明这些运算符的效果，我们打印出了二进制形式的值。图16.5中的程序打印了一个二进制形式的无符号整数，打印出的二进制数每8位编成一组。函数displayBits用按位与运算符操作变量value和displayMask。按位与运算符通常和一个称为“屏蔽字”（mask）的操作数一起使用。屏蔽字是指定位设置为1的整数值，用来在选择一个值的某些位时隐藏其他位。在函数displayBits中，屏蔽变量displayMask的值指定为1<<15（即10000000 00000000）。左移位运算符把displayMask中的值1从低位（最右边）移到高位（最左边），并用0填充右边的空位。语句：

```
cout << ( value & displayMask ? '1' : '0' );
```

确定变量value当前最左边位的打印值为1还是0。假定变量value的值为65000（即11111101 11101000），当用&操作value和displayMask时，因为值为0的位相应结果位的值也为0，所以除了高位外，变量value的所有其他位都会被屏蔽掉（隐藏）。如果最左边位为1，value & displayMask的值为10000000 00000000并打印出1，否则打印0。然后，表达式value<<=1把变量value向左移动1位（表达式value<<=1等价于value = value<<1）。变量value（unsigned类型）的每一位都循环这些步骤从而打印出了value的二进制值。图16.6列出了按位与运算符的操作结果。

```
1 // Fig. 16.5: fig16_05.cpp
2 // Printing an unsigned integer in bits
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 void displayBits( unsigned );
7
8 int main()
9 {
10     unsigned x;
11
12     cout << "Enter an unsigned integer: ";
13     cin >> x;
14     displayBits( x );
15     return 0;
```

```

16 }
17
18 void displayBits( unsigned value )
19 {
20     unsigned c, displayMask = 1 << 15;
21
22     cout << setw( 7 ) << value << " = ";
23
24     for ( c = 1; c <= 16; c++ ) {
25         cout << ( value & displayMask ? '1' : '0' );
26         value <<= 1;
27
28         if ( c % 8 == 0 )
29             cout << ' ';
30     }
31
32     cout << endl;
33 }

```

**输出结果:**

```

Enter an unsigned integer: 65000
65000 = 11111101 11101000

```

图 16.5 按位打印无符号整数

位 1	位 2	位 1&位 2
0	0	0
1	0	0
0	1	0
1	1	1

图 16.6 按位与运算符 & 对两个位的操作结果

**常见编程错误 16.5**

把逻辑与运算符 (&&) 和按位与运算符 (&) 相混淆。

图 16.7 中的程序演示了按位与、按位或、按位异或和按位取反运算符的用法。程序中用函数 displayBits 打印 unsigned 类型的整数值。输出结果见图 16.8。

```

1 // Fig. 16.7: fig16_07.cpp
2 // Using the bitwise AND, bitwise inclusive OR, bitwise
3 // exclusive OR, and bitwise complement operators.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 void displayBits( unsigned );
8
9 int main()
10 {
11     unsigned number1, number2, mask, setBits;
12
13     number1 = 65535;
14     mask = 1;
15     cout << "The result of combining the following\n";
16     displayBits( number1 );

```



```

17  displayBits( mask );
18  cout << "using the bitwise AND operator & is\n";
19  displayBits( number1 & mask );
20
21  number1 = 15;
22  setBits = 241;
23  cout << "\nThe result of combining the following\n";
24  displayBits( number1 );
25  displayBits( setBits );
26  cout << "using the bitwise inclusive OR operator | is\n";
27  displayBits( number1 | setBits );
28
29  number1 = 139;
30  number2 = 199;
31  cout << "\nThe result of combining the following\n";
32  displayBits( number1 );
33  displayBits( number2 );
34  cout << "using the bitwise exclusive OR operator ^ is\n";
35  displayBits( number1 ^ number2 );
36
37  number1 = 21845;
38  cout << "\nThe one's complement of\n";
39  displayBits( number1 );
40  cout << "is" << endl;
41  displayBits( ~number1 );
42
43  return 0;
44 }
45
46 void displayBits( unsigned value )
47 {
48     unsigned c, displayMask = 1 << 15;
49
50     cout << setw( 7 ) << value << " = ";
51
52     for ( c = 1; c <= 16; c++ ) {
53         cout << ( value & displayMask ? '1' : '0' );
54         value <<= 1;
55
56         if ( c % 8 == 0 )
57             cout << ' ';
58     }
59
60     cout << endl;
61 }

```

图 16.7 按位与、按位或、按位异或和按位取反运算符的用法

The result of combining the following  
 65535 = 11111111 11111111  
 1 = 00000000 00000001  
 using the bitwise AND operator & is  
 1 = 00000000 00000001

The result of combining the following  
 15 = 00000000 00001111  
 241 = 00000000 11110001

```
using the bitwise inclusive OR operator | is
255 = 00000000 11111111
```

The result of combining the following

```
139 = 00000000 10001011
```

```
199 = 00000000 11000111
```

```
using the bitwise exclusive OR operator ^ is
```

```
76 = 00000000 01001100
```

The one's complement of

```
21845 = 01010101 01010101
```

is

```
43690 = 10101010 10101010
```

图 16.8 图 16.7 中程序的运行结果

在图 16.7 中, 整数变量 `mask` 的赋值为 1 (即 00000000 00000001), 变量 `number1` 的赋值为 65535 (即 11111111 11111111)。表达式 `number1 & mask` 用按位与运算符 `&` 操作变量 `number1` 和 `mask`, 结果为 00000000 00000001。除了低位外, 变量 `number1` 的其他所有位都被变量 `mask` 屏蔽掉了。

按位或运算符用来把一个操作数中的指定位设置为 1。在图 16.7 的程序中, 变量 `number1` 的赋值为 15 (即 00000000 00001111), 变量 `setBits` 的赋值为 241 (即 00000000 11110001)。表达式 `number1 | setBits` 用按位或运算符操作变量 `number1` 和 `setBits`, 结果为 255 (即 00000000 11111111)。图 16.9 列出了按位或运算符对两个位的操作结果。

位 1	位 2	位 1 位 2
0	0	0
1	0	1
0	1	1
1	1	1

图 16.9 按位或运算符 (`|`) 对两个位的操作结果

#### 常见编程错误 16.6

把逻辑或运算符 (`||`) 和按位或运算符 (`|`) 相混淆。

按位异或运算符 (`^`) 在两个操作数相应位只有一个为 1 时把结果的相应位设置为 1。在图 16.7 的程序中, 变量 `number1` 和 `number2` 的赋值分别为 139 (即 00000000 10001011) 和 199 (即 00000000 11000111)。表达式 `number1 ^ number2` 用按位异或运算符操作了这两个变量, 结果为 00000000 01001100。图 16.10 列出了按位异或运算符对两个位的操作结果。

位 1	位 2	位 1 ^ 位 2
0	0	0
1	0	1
0	1	1
1	1	0

图 16.10 按位异或运算符 (`^`) 对两个位的操作结果

按位取反运算符 (`~`) 把操作数中的所有 1 位设置为 0、0 位设置为 1。在图 16.7 的程序中, 变量 `number1` 的赋值为 21845 (即 01010101 01010101), 表达式 `~number1` 的计算结果为 10101010 10101010。

图 16.11 中程序演示了左移位运算符 (<<) 和右移位运算符 (>>) 的用法。函数 displayBits 用来打印 unsigned 类型的整数值。

```

1 // Fig. 16.11: fig16_11.cpp
2 // Using the bitwise shift operators
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 void displayBits( unsigned );
7
8 int main()
9 {
10     unsigned number1 = 960;
11
12     cout << "The result of left shifting\n";
13     displayBits( number1 );
14     cout << "8 bit positions using the left "
15         << "shift operator is\n";
16     displayBits( number1 << 8 );
17     cout << "\nThe result of right shifting\n";
18     displayBits( number1 );
19     cout << "8 bit positions using the right "
20         << "shift operator is\n";
21     displayBits( number1 >> 8 );
22     return 0;
23 }
24
25 void displayBits( unsigned value )
26 {
27     unsigned c, displayMask = 1 << 15;
28
29     cout << setw( 7 ) << value << " = ";
30
31     for ( c = 1; c <= 16; c++ ) {
32         cout << ( value & displayMask ? '1' : '0' );
33         value <<= 1;
34
35         if ( c % 8 == 0 )
36             cout << ' ';
37     }
38
39     cout << endl;
40 }

```

**输出结果：**

```

The result of left shifting
  960 = 00000011 11000000
8 bit positions using the left shift operator << is
 49152 = 11000000 00000000

The result of right shifting
  960 = 00000011 11000000
8 bit positions using the right shift operator >> is
    3 = 00000000 00000011

```

图 16.11 移位运算符的用法

左移位运算符(<<)把其左操作数向左移动右操作数指定的位数,右边的空位用0填补,从左边移出的位值丢失了。在图 16.11 的程序中,变量 number1 的赋值为 960(即 00000011 11000000),表达式 number1<<8 把变量 number1 向左移动 8 位,结果为 49152(即 11000000 00000000)。

右移位运算符(>>)把其左操作数向右移动右操作数指定的位数。对 unsigned 类型的整数执行右移位操作使左边的空位用0填补,右边移出的值被丢失。在图 16.11 的程序中,表达式 number1>>8 将 number1 向右移动 8 位后的值为 3(即 00000000 00000011)。

#### 常见编程错误 16.7

如果右操作数是负值或右操作数大于左操作数存储的位数,移位操作的结果是不确定的。

#### 可移植性提示 16.4

右移位的结果与机器有关。右移位一个有符号整数,某些机器用0填补空位,还有一些机器用1填补空位。

每一种位运算符(除了按位取反运算符)都有一种相应的赋值运算符。图 16.12 列出这些位赋值运算符,这些运算符的用法与第 2 章介绍的算术赋值运算符用法类似。

位赋值运算符	
&=	按位与赋值运算符
=	按位或赋值运算符
^=	按位异或赋值运算符
<<=	左移位赋值运算符
>>=	右移位赋值运算符

图 16.12 位赋值运算符

图 16.13 列出了本书已经介绍过的各种运算符的优先级和结合律。这些运算符的优先级是按自顶向下降低的顺序排列的。

运算符	结合律	类型
:: (一元, 从右向左)      :: (二元, 从左向右)	从左到右	最高优先级
()    []    .    ->	从左到右	最高优先级
++    --    +    -    !    delete    sizeof	从右到左	一元运算符
*    &    new		
*    /    %	从左到右	乘法运算符
+    -	从左到右	加法运算符
<<    >>	从左到右	移位运算符
<    <=    >    >=	从左到右	关系运算符
==    !=	从左到右	相等运算符
&	从左到右	按位与运算符
^	从左到右	按位异或运算符
	从左到右	按位或运算符
&&	从左到右	逻辑与运算符
	从左到右	逻辑或运算符
?:	从右到左	条件运算符
=    +=    -=    *=    /=    %=	从右到左	赋值运算符
&=     =    ^=    <<=    >>=		
,	从左到右	逗号运算符

图 16.13 运算符的优先级和结合律

## 16.8 位段

对结构或类中的 `unsigned` 类型和 `int` 类型的成员，C 语言允许指定存储它们的位数，指定了存储位数的结构或类中的成员称为“位段”(bit field)。利用位段能够用最少的位数存储数据，从而更好地利用内存。位段成员必须声明为 `int` 或 `unsigned` 类型。

### 性能提示 16.2

利用位段能够节省内存。

考虑如下结构定义：

```
struct BitCard {
    unsigned face : 4;
    unsigned suit : 2;
    unsigned color : 1;
};
```

该定义中包含了三个 `unsigned` 类型的位段：`face`、`suit` 和 `color`，这三个位段代表了 52 张牌中的某张牌。声明位段的方法是：在 `unsigned` 或 `int` 类型的成员名后依次加上冒号(:)和代表位段宽度的整数常量(位段宽度就是存储该成员的位数)。代表位段宽度的常量必须是在 0 到系统用来存储 `int` 类型值的位数之间的整数值。本书中的范例是在 2 字节整数(16 位)的计算机上测试的。

上述结构的定义表示成员 `face` 的存储空间占 4 位、成员 `suit` 的存储空间占 2 位、成员 `color` 的存储空间占 1 位。位数是根据每一个成员所需的取值范围确定的。成员 `face` (牌的面值) 的存储值在 0 (Ace) 到 12 (King) 之间，所以让它占用 4 位(4 位能够存储 0 到 15 之间的值)。成员 `suit` (牌的花色) 的存储值在 0 到 3 之间(0 = Diamonds、1 = Hearts、2 = Clubs、3 = Spades)，所以让它占用 2 位(2 位能够存储 0 到 3 之间的值)。成员 `color` (系统允许显示的花色的红黑属性) 存储 0 (Red) 或存储 1 (Black)，所以用 1 位就可以了。

图 16.14 中的程序(输出见图 16.15)建立了有 52 个元素的 `struct bitCard` 类型的数组 `deck`。函数 `fillDeck` 在数组 `deck` 中插入 52 张牌，函数 `deal` 打印 52 张牌。可以看到，访问结构的位段成员方法同访问其他结构成员的方法是一样的。在结构中包含成员 `color` 是用来表示系统允许显示的牌面颜色。

```
1 // Fig. 16.14: fig16_14.cpp
2 // Example using a bit field
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 struct BitCard {
7     unsigned face : 4;
8     unsigned suit : 2;
9     unsigned color : 1;
10 };
11
12 void fillDeck( BitCard * );
13 void deal( BitCard * );
14
15 int main()
16 {
```

```

17  BitCard deck[ 52 ];
18
19  fillDeck( deck );
20  deal( deck );
21  return 0;
22 }
23
24 void fillDeck( BitCard *wDeck )
25 {
26     for ( int i = 0; i <= 51; i++ ) {
27         wDeck[ i ].face = i % 13;
28         wDeck[ i ].suit = i / 13;
29         wDeck[ i ].color = i / 26;
30     }
31 }
32
33 // Output cards in two column format. Cards 0-25 subscripted
34 // with k1 (column 1). Cards 26-51 subscripted k2 in (column 2.)
35 void deal( BitCard *wDeck )
36 {
37     for ( int k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
38         cout << "Card:" << setw( 3 ) << wDeck[ k1 ].face
39             << " Suit:" << setw( 2 ) << wDeck[ k1 ].suit
40             << " Color:" << setw( 2 ) << wDeck[ k1 ].color
41             << " " << "Card:" << setw( 3 ) << wDeck[ k2 ].face
42             << " Suit:" << setw( 2 ) << wDeck[ k2 ].suit
43             << " Color:" << setw( 2 ) << wDeck[ k2 ].color
44             << endl;
45     }
46 }

```

图 16.14 用位段存储一副牌

Card: 0	Suit: 0	Color: 0	Card: 0	Suit: 2	Color: 1
Card: 1	Suit: 0	Color: 0	Card: 1	Suit: 2	Color: 1
Card: 2	Suit: 0	Color: 0	Card: 2	Suit: 2	Color: 1
Card: 3	Suit: 0	Color: 0	Card: 3	Suit: 2	Color: 1
Card: 4	Suit: 0	Color: 0	Card: 4	Suit: 2	Color: 1
Card: 5	Suit: 0	Color: 0	Card: 5	Suit: 2	Color: 1
Card: 6	Suit: 0	Color: 0	Card: 6	Suit: 2	Color: 1
Card: 7	Suit: 0	Color: 0	Card: 7	Suit: 2	Color: 1
Card: 8	Suit: 0	Color: 0	Card: 8	Suit: 2	Color: 1
Card: 9	Suit: 0	Color: 0	Card: 9	Suit: 2	Color: 1
Card: 10	Suit: 0	Color: 0	Card: 10	Suit: 2	Color: 1
Card: 11	Suit: 0	Color: 0	Card: 11	Suit: 2	Color: 1
Card: 12	Suit: 0	Color: 0	Card: 12	Suit: 2	Color: 1
Card: 0	Suit: 1	Color: 0	Card: 0	Suit: 3	Color: 1
Card: 1	Suit: 1	Color: 0	Card: 1	Suit: 3	Color: 1
Card: 2	Suit: 1	Color: 0	Card: 2	Suit: 3	Color: 1
Card: 3	Suit: 1	Color: 0	Card: 3	Suit: 3	Color: 1
Card: 4	Suit: 1	Color: 0	Card: 4	Suit: 3	Color: 1
Card: 5	Suit: 1	Color: 0	Card: 5	Suit: 3	Color: 1
Card: 6	Suit: 1	Color: 0	Card: 6	Suit: 3	Color: 1
Card: 7	Suit: 1	Color: 0	Card: 7	Suit: 3	Color: 1
Card: 8	Suit: 1	Color: 0	Card: 8	Suit: 3	Color: 1
Card: 9	Suit: 1	Color: 0	Card: 9	Suit: 3	Color: 1

Card: 10	Suit: 1	Color: 0	Card: 10	Suit: 3	Color: 1
Card: 11	Suit: 1	Color: 0	Card: 11	Suit: 3	Color: 1
Card: 12	Suit: 1	Color: 0	Card: 12	Suit: 3	Color: 1

图 16.15 图 16.14 程序的输出结果

可以指定内存不使用的无名位段用作结构中的填充。例如，定义了占用3位的无名位段，这3位不存储任何内容。这样，在字长为2个字节的计算机上，成员b就被存储到了另一个内存单元中。例如，结构定义：

```
struct Example {
    unsigned a : 13;
    unsigned   : 3;
    unsigned b : 4;
};
```

0宽度的无名位段用来使下一个位段从新内存单元的起始位置开始存储。例如，结构定义：

```
struct Example {
    unsigned a : 13;
    unsigned   : 0;
    unsigned b : 4;
};
```

用0长度的无名位段跳过存储a的内存单元中剩下的位，并把b存储到下一个内存单元中（从起始位置开始存储）。

#### 可移植性提示 16.5

位段操作是与机器有关的。例如，某些计算机允许位段跨越字边界，而另一些计算机却不允许这样做。

#### 常见编程错误 16.8

试图像访问数组元素一样访问位段中的某个位。位段不是位数组。

#### 常见编程错误 16.9

试图取得位段的地址。因为位段没有地址，所以不能对位段使用运算符&。

#### 性能提示 16.3

尽管使用位段能够节省内存，但是会使编译器产生执行速度慢的机器语言代码。这是因为只访问已编址内存单元的某部分需要机器语言完成额外的操作。这是计算机科学中的“时间空间”（速度与内存）冲突的一个例子。

## 16.9 字符处理库

大多数数据都是以字符形式输入计算机的，包括字母、数字和各种特殊字符。本节介绍C++如何检查和操作各个字符。本章余下部分要继续讨论第5章开始介绍的字符串操作。

字符处理库包括几个测试和操作字符数据的函数。每个函数接收一个字符，表示为int类型，或接受一个EOF参数。字符通常以整数形式进行操作。记住，EOF通常取-1，有些硬件结构不允许在char变量中存放负值。因此，字符处理函数将字符作为整数操作。图16.16总结了字符处理函数库中的函数。在使用字符处理函数库中的函数时，要在程序中包括头文件<ctype.h>。

函数原型	函数描述
<code>int isdigit( int c )</code>	如果 <code>c</code> 是一个数字, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>int isalpha( int c )</code>	如果 <code>c</code> 是一个字母, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>int isalnum( int c )</code>	如果 <code>c</code> 是一个数字或字母, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>int isxdigit( int c )</code>	如果 <code>c</code> 是一个十六进制数字字符, 返回 <code>true</code> , 否则返回 <code>false</code> (二进制数、八进制数、十进制数和十六进制数的解释见附录 C “数值系统”)
<code>int islower( int c )</code>	如果 <code>c</code> 是一个小写字母, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>int isupper( int c )</code>	如果 <code>c</code> 是一个大写字母, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>int tolower( int c )</code>	如果 <code>c</code> 是一个大写字母, 返回其小写字母, 否则返回原参数
<code>int toupper( int c )</code>	如果 <code>c</code> 是一个小写字母, 返回其大写字母, 否则返回原参数
<code>int isspace( int c )</code>	如果 <code>c</code> 是一个空白字符, 返回 <code>true</code> , 否则返回 <code>false</code> 。空白字符是指换行符 (‘\n’)、空格 (‘ ’)、进纸符 (‘\f’)、回车 (‘\r’)、水平制表符 (‘\t’) 和垂直制表符 (‘\v’)
<code>int iscntrl( int c )</code>	如果 <code>c</code> 是一个控制符, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>int ispunct( int c )</code>	如果 <code>c</code> 是一个除空格、数字和字母之外的可打印字符, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>int isprint( int c )</code>	如果 <code>c</code> 是一个可打印字符 (包括空格), 返回 <code>true</code> , 否则返回 <code>false</code>
<code>int isgraph( int c )</code>	如果 <code>c</code> 是除空格之外的可打印字符, 返回 <code>true</code> , 否则返回 <code>false</code>

图 16.16 字符处理库函数小结

图 16.17 中的程序演示了函数 `isdigit`、`isalpha`、`isalnum` 和 `isxdigit` 的用法。函数 `isdigit` 确定其参数是否是一个 0~9 之间的数字。函数 `isalpha` 确定其参数是否是一个 A~Z 的大写字母或 a~z 的小写字母。函数 `isalnum` 确定其参数是否是一个大写字母、一个小写字母或一个数字。函数 `isxdigit` 确定其参数是否是一个十六进制数字 (A~F、a~f、0~9)。

```

1 // Fig. 16.17: fig16_17.cpp
2 // Using functions isdigit, isalpha, isalnum, and isxdigit
3 #include <iostream.h>
4 #include <ctype.h>
5
6 int main()
7 {
8     cout << "According to isdigit:\n"
9         << ( isdigit( '8' ) ? "8 is a" : "8 is not a" )
10        << " digit\n"
11        << ( isdigit( '#' ) ? "# is a" : "# is not a" )
12        << " digit\n";
13     cout << "\nAccording to isalpha:\n"
14         << ( isalpha( 'A' ) ? "A is a" : "A is not a" )
15         << " letter\n"
16         << ( isalpha( 'b' ) ? "b is a" : "b is not a" )
17         << " letter\n"
18         << ( isalpha( '&' ) ? "& is a" : "& is not a" )
19         << " letter\n"
20         << ( isalpha( '4' ) ? "4 is a" : "4 is not a" )
21         << " letter\n";
22     cout << "\nAccording to isalnum:\n"
23         << ( isalnum( 'A' ) ? "A is a" : "A is not a" )
24         << " digit or a letter\n"
25         << ( isalnum( '8' ) ? "8 is a" : "8 is not a" )
26         << " digit or a letter\n"
27         << ( isalnum( '#' ) ? "# is a" : "# is not a" )
28         << " digit or a letter\n";

```



```

29     cout << "\nAccording to isxdigit:\n"
30         << ( isxdigit( 'F' ) ? "F is a" : "F is not a" )
31         << " hexadecimal digit\n"
32         << ( isxdigit( 'J' ) ? "J is a" : "J is not a" )
33         << " hexadecimal digit\n"
34         << ( isxdigit( '7' ) ? "7 is a" : "7 is not a" )
35         << " hexadecimal digit\n"
36         << ( isxdigit( '$' ) ? "$ is a" : "$ is not a" )
37         << " hexadecimal digit\n"
38         << ( isxdigit( 'f' ) ? "f is a" : "f is not a" )
39         << " hexadecimal digit" << endl;
40     return 0;
41 }

```

**输出结果:**

According to isdigit:

8 is a digit

# is not a digit

According to isalpha:

A is a letter

b is a letter

& is not a letter

4 is not a letter

According to isalnum:

A is a digit or a letter

8 is a digit or a letter

# is not a digit or a letter

According to isxdigit:

F is a hexadecimal digit

J is not a hexadecimal digit

7 is a hexadecimal digit

\$ is not a hexadecimal digit

f is a hexadecimal digit

图 16.17 函数 isdigit、isalpha、isalnum 和 isxdigit 的用法

图 16.17 中的程序用条件运算符 (?:) 确定每一个函数在测试完字符后是打印出 “is a” 还是 “is not a”。例如, 表达式:

```
isdigit( '8' ) ? "8 is a" : "8 is not a"
```

的意思是: 如果 '8' 是一个数字 (即 isdigit 返回非 0 的值), 打印字符串 "8 is a", 否则 (即 isdigit 返回 0) 打印字符串 "8 is not a"。

图 16.18 中的程序演示了函数 islower、isupper、tolower 和 toupper 的用法。函数 islower 确定其参数是否是一个小写字母 (a~z)。函数 isupper 确定其参数是否一个大写字母 (A~Z)。函数 tolower 把大写字母转换为小写字母并返回该小写字母, 如果该参数不是一个大写字母, tolower 返回原参数。函数 toupper 把小写字母转换为大写字母并返回该大写字母, 如果该参数不是一个小写字母, toupper 返回原参数。

```

1 // Fig. 16.18: fig16_18.cpp
2 // Using functions islower, isupper, tolower, toupper
3 #include <iostream.h>
4 #include <ctype.h>
5
6 int main()
7 {
8     cout << "According to islower:\n"
9         << ( islower( 'p' ) ? "p is a" : "p is not a" )
10        << " lowercase letter\n"
11        << ( islower( 'P' ) ? "P is a" : "P is not a" )
12        << " lowercase letter\n"
13        << ( islower( '5' ) ? "5 is a" : "5 is not a" )
14        << " lowercase letter\n"
15        << ( islower( '!' ) ? "! is a" : "! is not a" )
16        << " lowercase letter\n";
17     cout << "\nAccording to isupper:\n"
18         << ( isupper( 'D' ) ? "D is an" : "D is not an" )
19         << " uppercase letter\n"
20         << ( isupper( 'd' ) ? "d is an" : "d is not an" )
21         << " uppercase letter\n"
22         << ( isupper( '8' ) ? "8 is an" : "8 is not an" )
23         << " uppercase letter\n"
24         << ( isupper( '$' ) ? "$ is an" : "$ is not an" )
25         << " uppercase letter\n";
26     cout << "\nu converted to uppercase is "
27         << ( char ) toupper( 'u' )
28         << "\n7 converted to uppercase is "
29         << ( char ) toupper( '7' )
30         << "\n$ converted to uppercase is "
31         << ( char ) toupper( '$' )
32         << "\nL converted to lowercase is "
33         << ( char ) tolower( 'L' ) << endl;
34
35     return 0;
36 }

```

#### 输出结果:

```

According to islower:
P is a lowercase letter
p is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

```

```

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

```

```

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to uppercase is l

```

图 16.18 函数 islower、isupper、tolower 和 toupper 的用法

图 16.19 中的程序演示了函数 `isspace`、`isctrl`、`ispunct`、`isprint` 和 `isgraph` 的用法。函数 `isspace` 确定其参数是否是如下空白字符之一：空格（' '）、进纸符（'\f'）、换行符（'\n'）、回车（'\r'）、水平制表符（'\t'）或垂直制表符（'\v'）。函数 `isctrl` 确定其参数是否是如下几个控制符之一：水平制表符（'\t'）、垂直制表符（'\v'）、进纸符（'\f'）、响铃符（'\a'）、退格符（'\b'）、回车符（'\r'）和换行符（'\n'）。函数 `ispunct` 确定其参数是否是除空格、数字和字母之外的可打印字符，如 \$、#、(、)、[、]、{、}、:、;、% 等等。函数 `isprint` 确定其参数是否是一个可显示在屏幕上的字符（包括空格）。函数 `isgraph` 和函数 `isprint` 测试同样的字符数据，但不包括空格。

```

1 // Fig. 16.19: fig16_19.cpp
2 // Using functions isspace, isctrl, ispunct, isprint, isgraph
3 #include <iostream.h>
4 #include <ctype.h>
5
6 int main()
7 {
8     cout << "According to isspace:\nNewline "
9         << ( isspace( '\n' ) ? "is a" : "is not a" )
10        << " whitespace character\nHorizontal tab "
11        << ( isspace( '\t' ) ? "is a" : "is not a" )
12        << " whitespace character\n"
13        << ( isspace( '%' ) ? "% is a" : "% is not a" )
14        << " whitespace character\n";
15     cout << "\nAccording to isctrl:\nNewline "
16         << ( isctrl( '\n' ) ? "is a" : "is not a" )
17         << " control character\n"
18         << ( isctrl( '$' ) ? "$ is a" : "$ is not a" )
19         << " control character\n";
20     cout << "\nAccording to ispunct:\n"
21         << ( ispunct( ';' ) ? "; is a" : "; is not a" )
22         << " punctuation character\n"
23         << ( ispunct( 'Y' ) ? "Y is a" : "Y is not a" )
24         << " punctuation character\n"
25         << ( ispunct( '#' ) ? "# is a" : "# is not a" )
26         << " punctuation character\n";
27     cout << "\nAccording to isprint:\n"
28         << ( isprint( '$' ) ? "$ is a" : "$ is not a" )
29         << " printing character\nAlert "
30         << ( isprint( '\a' ) ? "is a" : "is not a" )
31         << " printing character\n";
32     cout << "\nAccording to isgraph:\n"
33         << ( isgraph( 'Q' ) ? "Q is a" : "Q is not a" )
34         << " printing character other than a space\nSpace "
35         << ( isgraph( ' ' ) ? "is a" : "is not a" )
36         << " printing character other than a space" << endl;
37
38     return 0;
39 }

```

#### 输出结果:

```

According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

```

```

According to ispunct:
; is punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space

```

图 16.19 函数 isspace、iscntrl、ispunct、isprint 和 isgraph 的用法

## 16.10 字符串转换函数

第5章介绍了C++最常用的字符串操作函数。下面几节要介绍其他函数，包括将字符串变为数字值的函数、查找字符串的函数和操作、比较与查找内存块的函数。

本节介绍通用实用程序库（stdlib）中的字符串转换函数。这些函数把数字字符串转换为整数和浮点数值。图 16.20 列出字符串转换函数。函数首部用 const 声明了变量 nPtr（读法：nPtr 是一个指向字符常量的指针），const 说明参数 nPtr 值是不可修改的。在使用通用实用程序库中的函数时，程序要包含 <stdlib.h> 头文件。

函数原型	函数描述
double atof( const char *nPtr )	把字符串 nPtr 转换为 double 类型
int atoi( const char *nPtr )	把字符串 nPtr 转换为 int 类型
long atol( const char *nPtr )	把字符串 nPtr 转换为 long int 类型
double strtod( const char *nPtr, char **endPtr )	把字符串 nPtr 转换为 double 类型
long strtol( const char *nPtr, char **endPtr, int base )	把字符串 nPtr 转换为 long 类型
unsigned long strtoul( const char *nPtr, char **endPtr, int base )	把字符串 nPtr 转换为 unsigned long 类型

图 16.20 通用实用程序库中的字符串转换函数

函数 atof（见图 16.21）把其参数（代表浮点数值字符串）转换为 double 类型的值，并返回该 double 类型的值。如果不能把参数转换为一个浮点数（例如，字符串的第一个字符不是一个数字），那么函数 atof 返回 0。

```

1 // Fig. 16.21: fig16_21.cpp
2 // Using atof
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     double d = atof( "99.0" );
9
10    cout << "The string \"99.0\" converted to double is "
11         << d << "\nThe converted value divided by 2 is "
12         << d / 2.0 << endl;

```

```
13     return 0;
14 }
```

**输出结果:**

```
The string "99.0" converted to double is 99
The converted value divided by 2 is 49.5
```

图 16.21 函数 atof 的用法

函数 atoi (见图 16.22) 把其参数 (代表一个整数的数字字符串) 转换为一个 int 类型的值, 并返回该 int 类型的值。如果不能表示转换的值, 函数 atoi 返回 0。

```
1 // Fig. 16.22: fig16_22.cpp
2 // Using atoi
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main( )
7 {
8     int i = atoi( "2593" );
9
10    cout << "The string \"2593\" converted to int is " << i
11         << "\nThe converted value minus 593 is " << i - 593
12         << endl;
13    return 0;
14 }
```

**输出结果:**

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

图 16.22 函数 atoi 的用法

函数 atol (见图 16.23) 把其参数 (代表长整数的一个数字字符串) 转换为一个 long 类型的值, 并返回该 long 类型的值。如果转换不成功, 函数 atol 返回 0。如果 int 和 long 类型的值都是用 4 个字节存储的, 那么函数 atoi 和函数 atol 是等价的。

```
1 // Fig. 16.23: fig16_23.cpp
2 // Using atol
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     long l = atol( "1000000" );
9
10    cout << "The string \"1000000\" converted to long is " << l
11         << "\nThe converted value divided by 2 is " << l / 2
12         << endl;
13    return 0;
14 }
```

**输出结果:**

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

图 16.23 函数 atol 的用法

函数 `strtod` (见图 16.24) 把一个代表浮点数的字符序列转换为 `double` 类型的值。函数有两个参数：一个字符串 (`char*`) 和一个指向字符串的指针。字符串中包含了要转换为 `double` 类型值的字符序列，指针指向字符串中已转换部分之后的第一个字符的内存单元。图 16.24 程序中的语句：

```
d = strtod( string, &stringPtr );
```

表示把从 `string` 转换而来的 `double` 类型的值赋给 `d`，把 `string` 中已转换值的部分 (51.2) 之后的第一个字符的内存单元赋给 `&stringPtr`。

```
1 // Fig. 16.24: fig16_24.cpp
2 // Using strtod
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     double d;
9     char *string = "51.2% are admitted", *stringPtr;
10
11     d = strtod( string, &stringPtr );
12     cout << "The string \"" << string
13         << "\" is converted to the\ndouble value " << d
14         << " and the string \"" << stringPtr << "\" " << endl;
15     return 0;
16 }
```

**输出结果：**

```
The string "51.2% are admitted" is converted to the
double value 51.2 and the string "% are admitted"
```

图 16.24 函数 `strtod` 的用法

函数 `strtol` (见图 16.25) 把代表某个整数的一个字符序列转换为 `long` 类型。函数有三个参数：一个字符串 (`char*`)、一个指向字符串的指针和一个整数。字符串中包含了要转换的字符序列，指针指向字符串中已转换部分之后的第一个字符的内存单元，整数指定了转换值的基数。图 16.25 程序中的语句：

```
x = strtol( string, &remainderPtr, 0);
```

表示把从 `string` 转换而来的 `long` 类型的值赋给 `x`，`string` 中已转换部分之后的剩余部分赋给第二个参数 `&remainderPtr`。若第二个参数为 `NULL` 则忽略字符串中的剩余部分。第三个参数 0 表示要被转换的值可以是八进制 (基数为 8)、十进制 (基数为 10) 或十六进制 (基数为 16) 格式。基数可指定为 0 或 2 到 36 之间的任何值。八进制、十进制和十六进制数值系统的详细介绍参见附录“数值系统”。基数从 11 到 36 的整数值表示用字符 A~Z 代表值 10 到 35。例如，十六进制值可包含数字 0~9 和字符 A~F，基数为 11 的整数可包含数字 0~9 和字符 A，基数为 24 的整数可包含数字 0~9 和字符 A~N，基数为 36 的整数可包含数字 0~9 和字符 A~Z。

```

1 // Fig. 16.25: fig16_25.cpp
2 // Using strtol
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     long x;
9     char *string = "-1234567abc", *remainderPtr;
10
11     x = strtol( string, &remainderPtr, 0 );
12     cout << "The original string is \"" << string
13         << "\"\n" << "The converted value is " << x
14         << "\"\n" << "The remainder of the original string is \""
15         << remainderPtr
16         << "\"\n" << "The converted value plus 567 is "
17         << x + 567 << endl;
18     return 0;
19 }

```

**输出结果:**

```

The original string is "-1233567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000

```

图 16.25 函数 strtol 的用法

函数 strtoul ( 见图 16.26 ) 把代表某个 unsigned long 类型的整数值的字符序列转换为 unsigned long 类型。该函数和 strtol 的用法相同。图 16.26 程序中的语句:

```
x = strtoul( string, &remainderPtr, 0 );
```

表示把从 string 转换而来的 unsigned long 类型的值赋给 x, 把 string 中转换部分之后的剩余部分赋给第二个参数 &remainderPtr, 第三个参数 0 表示要被转换的值可以是八进制、十进制和十六进制格式。

```

1 // Fig. 16.26: fig16_26.cpp
2 // Using strtoul
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     unsigned long x;
9     char *string = "1234567abc", *remainderPtr;
10
11     x = strtoul( string, &remainderPtr, 0 );
12     cout << "The original string is \"" << string
13         << "\"\n" << "The converted value is " << x
14         << "\"\n" << "The remainder of the original string is \""
15         << remainderPtr
16         << "\"\n" << "The converted value minus 567 is "
17         << x - 567 << endl;

```

```

18     return 0;
19 }

```

**输出结果：**

```

The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000

```

图 16.26 函数 `strtoul` 的用法

## 16.11 字符串处理库中的查找函数

本节介绍字符串处理库中用来在字符串中查找字符或其他字符串的函数。图 16.27 列出了这些函数。函数 `strcspn` 和 `strspn` 的返回类型是 `size_t`。`size_t` 是运算符 `sizeof` 返回值的整数类型。

**可移植性提示 16.6**

类型 `size_t` 与系统有关，它或者是 `unsigned long` 类型，或者是 `unsigned int` 类型。

函数原型	函数描述
<code>char* strchr( const char*s, int c )</code>	定位字符 <code>c</code> 首次出现在字符串 <code>s</code> 中的位置。如果找到了 <code>c</code> ，返回一个指向 <code>s</code> 中字符 <code>c</code> 的指针，否则返回 <code>NULL</code> 指针
<code>size_t strcspn( const char*s1, const char*s2 )</code>	计算并返回字符串 <code>s1</code> 中不含字符串 <code>s2</code> 中字符的起始段的长度
<code>size_t strspn( const char*s1, const char*s2 )</code>	计算并返回字符串 <code>s1</code> 中只含字符串 <code>s2</code> 中字符的起始段的长度
<code>char* strpbrk( const char*s1, const char*s2 )</code>	定位字符串 <code>s1</code> 首次出现在字符串 <code>s2</code> 中字符的位置。如果找到了字符串 <code>s2</code> 中的字符，返回一个指向字符串 <code>s1</code> 中该字符的指针，否则返回 <code>NULL</code> 指针
<code>char* strrchr( const char*s, int c )</code>	定位字符串 <code>s</code> 中最后一次出现 <code>c</code> 的位置。如果找到了 <code>c</code> ，返回一个指向字符串 <code>s</code> 中字符 <code>c</code> 的指针，否则返回 <code>NULL</code> 指针
<code>char* strstr( const char*s1, const char*s2 )</code>	定位在字符串 <code>s1</code> 中首次出现字符串 <code>s2</code> 的位置。如果找到了该字符串，返回一个指向 <code>s1</code> 中的该字符串的指针，否则返回 <code>NULL</code> 指针

图 16.27 字符串处理库中的字符串操作函数

函数 `strchr` 在一个字符串中查找第一次出现的某个字符。如果找到了该字符，返回一个指向字符串中该字符的指针，否则 `strchr` 返回 `NULL`。图 16.28 中的程序用 `strchr` 函数查找首次出现在字符串 "This is a test" 中的字符 'a' 和 'z'。

```

1 // Fig. 16.28: fig16_28.cpp
2 // Using strchr
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {

```



```

8   char *string = "This is a test";
9   char character1 = 'a', character2 = 'z';
10
11   if ( strchr( string, character1 ) != NULL )
12       cout << '\'' << character1 << "' was found in \""
13           << string << "\".\n";
14   else
15       cout << '\'' << character1 << "' was not found in \""
16           << string << "\".\n";
17
18   if ( strchr( string, character2 ) != NULL )
19       cout << '\'' << character2 << "' was found in \""
20           << string << "\".\n";
21   else
22       cout << '\'' << character2 << "' was not found in \""
23           << string << "\"." << endl;
24   return 0;
25 }

```

**输出结果:**

```

'a' was found in "This is a test"
'z' was not found in "This is a test".

```

图 16.28 函数 strchr 的用法

函数 `strcspn` (见图 16.29) 计算其第一个字符串参数中不含第二个字符串参数中字符的起始段的长度, 并返回该长度。

```

1 // Fig. 16.29: fig16_29.cpp
2 // Using strcspn
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "The value is 3.14159";
9     char *string2 = "1234567890";
10
11     cout << "string1 = " << string1 << "\nstring2 = " << string2
12         << "\n\nThe length of the initial segment of string1"
13         << "\n\ncontaining no characters from string2 = "
14         << strcspn( string1, string2 ) << endl;
15     return 0;
16 }

```

**输出结果:**

```

string1 = The value is 3.14159
string2 = 1234567890

```

```

The length of the initial segment of string1
containing no characters from string2 = 13

```

图 16.29 函数 strcspn 的用法

函数 `strpbrk` 在其第一个字符串参数中查找首次出现第二个字符串参数中字符的位置。如果找到了第二个参数中的字符, 返回一个指向第一个参数中该字符的指针, 否则返回 `NULL`。图 16.30 中的程序定位在 `string1` 中首次出现的 `string2` 中的字符。

```
1 // Fig. 16.30: fig16_30.cpp
2 // Using strpbrk
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "This is a test";
9     char *string2 = "beware";
10
11     cout << "Of the characters in \"" << string2 << "\"\n"
12         << *strpbrk( string1, string2 ) << '\n'
13         << " is the first character to appear in\n\"
14         << string1 << '\"' << endl;
15     return 0;
16 }
```

**输出结果:**

```
Of the characters in "beware"
'a' is the first character to appear in
"This is a test"
```

图 16.30 函数 `strpbrk` 的用法

函数 `strrchr` 在某个字符串中查找最后一次出现指定字符的位置。如果找到该字符, 返回一个指向字符串中的该字符的指针, 否则返回 `NULL`。图 16.31 中的程序在字符串 "A zoo has many animals including zebras" 中查找最后一次出现字符 'z' 的位置。

```
1 // Fig. 16.31: fig16_31.cpp
2 // Using strrchr
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "A zoo has many animals including zebras";
9     int c = 'z';
10
11     cout << "The remainder of string1 beginning with the\n"
12         << "last occurrence of character '" << (char) c
13         << "' is: \"" << strrchr( string1, c ) << '\"' << endl;
14     return 0;
15 }
```

**输出结果:**

```
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

图 16.31 函数 `strrchr` 的用法

函数 `strspn` (见图16.32) 计算第一个字符串参数中只包含第二个字符串参数中字符的起始段的长度, 并返回该长度。

```

1 // Fig. 16.32: fig16_32.cpp
2 // Using strspn
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "The value is 3.14159";
9     char *string2 = "aehilsTuv ";
10
11     cout << "string1 = " << string1
12         << "\nstring2 = " << string2
13         << "\n\nThe length of the initial segment of string1\n"
14         << "containing only characters from string2 = "
15         << strspn( string1, string2 ) << endl;
16     return 0;
17 }

```

**输出结果:**

```

string1 = The value is 3.14159
string2 = aehilsTuv

```

```

The length of the initial segment of string1
containing only characters from string2 = 13

```

图 16.32 函数 `strspn` 的用法

函数 `strstr` 在第一个字符串参数中查找首次出现其第二个字符串参数的位置。如果在第一个字符串中找到了第二个字符串, 返回一个指向在第一个参数中的该字符串的指针。图 16.33 中的程序用函数 `strstr` 在字符串 "abcdefabcdef" 中查找字符串 "def"。

```

1 // Fig. 16.33: fig16_33.cpp
2 // Using strstr
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "abcdefabcdef";
9     char *string2 = "def";
10
11     cout << "string1 = " << string1 << "\nstring2 = " << string2
12         << "\n\nThe remainder of string1 beginning with the\n"
13         << "first occurrence of string2 is: "
14         << strstr( string1, string2 ) << endl;
15     return 0;
16 }

```

**输出结果:**

```

string1 = abcdefabcdef

```

```
string2 = def
```

```
The remainder of string1 beginning with the
First occurrence of string2 is: defabcdef
```

图 16.33 函数 strstr 的用法

## 16.12 字符串处理库中的内存函数

本节介绍的字符串处理库中的函数用来操作、比较和查找内存块。这些函数像处理数组一样处理内存块，他们能够操作任何数据块。图 16.34 列出了字符串处理库的内存函数。在讨论函数时，“对象”是指数据块。

函数原型	函数描述
<code>void* memcpy( void*s1, const void*s2, size_t n )</code>	把s2所指向的对象中的n个字符复制到s1所指向的对象中，返回指向结果对象的指针
<code>void* memmove( void*s1, const void*s2, size_t n )</code>	把s2所指向的对象中的n个字符复制到s1所指向的对象中。复制过程就好像是先把字符从s2所指向的对象复制到临时数组中，然后再从临时数组复制到s1所指向的对象中。返回指向结果对象的指针
<code>int memcmp( const void*s1, const void*s2, size_t n )</code>	比较s1和s2所指向对象的前n个字符。如果s1所指向对象中的字符等于、小于或大于s2所指向对象中的字符，返回值分别等于0、小于0和大于0
<code>void* memchr( const void*s, int c, size_t n )</code>	定位在s所指向对象的前n个字符中首次出现字符c（转换为unsigned char类型）的位置。如果找到c，返回指向它的指针，否则返回0
<code>void* memset( void*s, int c, size_t n )</code>	把c（转换为unsigned char类型）复制到s所指向的对象的前n个字符中。返回指向结果的指针

图 16.34 字符串处理库中的内存函数

这些函数的指针参数都被声明为void\*类型。从第5章的内容可以知道，指向任何数据类型的指针都可以直接赋给void\*类型的指针，因此，这些函数能够接收指向任何数据类型的指针。因为不能复引用void\*类型的指针，所以每一个函数都接收一个说明函数要处理的字符个数（字节数）的参数。为简单起见，本节中的例子对字符数组（字符块）进行处理。

函数memcpy把指定个数的字符从第二个指针参数所指向的对象复制到第一个指针参数所指向的对象。该函数可接收指向任何对象的指针。如果两个对象在内存中重叠（即有一部分是相同的），函数的返回结果是不确定的。图16.35中的程序用函数memcpy把数组s2中的字符串复制到数组s1中。

```
1 // Fig. 16.35: fig16_35.cpp
2 // Using memcpy
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char s1[ 17 ], s2[] = "Copy this string";
9 }
```

```
10    memcpy( s1, s2, 17 );
11    cout << "After s2 is copied into s1 with memcpy,\n"
12          << "s1 contains \"" << s1 << "\"" << endl;
13    return 0;
14 }
```

**输出结果:**

```
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
```

图 16.35 函数 memcpy 的用法

与函数memcpy类似,函数memmove把指定个数的字节数从第二个指针参数所指向的对象复制到第一个指针参数所指向的对象中。复制过程就好像是先把指定个数的字节数从第二个指针参数所指向的对象复制到一个临时字符数组中,然后再从临时数组复制到第一个指针参数所指向的对象中。这种复制过程能够把字符串中的某一部分复制到同一个字符串的另一部分中。

**常见编程错误 16.10**

除memmove之外的字符串操作函数在同一字符串中的不同部分相互复制字符时,其结果是不确定的。

图 16.36 中的程序用 memmove 函数把数组 x 的最后 10 个字节复制到数组 x 的前 10 个字节中。

```
1 // Fig. 16.36: fig16_36.cpp
2 // Using memmove
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char x[] = "Home Sweet Home";
9
10    cout << "The string in array x before memmove is: " << x;
11    cout << "\nThe string in array x after memmove is: "
12          << (char *) memmove( x, &x[ 5 ], 10 ) << endl;
13    return 0;
14 }
```

**输出结果:**

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is: Sweet Home Home
```

图 16.36 函数 memmove 的用法

函数 memcmp (见图 16.37) 把第一个参数中指定个数的字符与第二参数中的相应的字符进行比较。如果第一个参数中的字符大于第二参数中的字符,返回一个大于0的值。如果第一个参数的字符等于第二个参数的字符,返回值等于0。如果第一个参数的字符小于第二个参数的字符,返回一个小于0的值。

```
1 // Fig. 16.37: fig16_37.cpp
2 // Using memcmp
3 #include <iostream.h>
4 #include <iomanip.h>
```

```

5 #include <string.h>
6
7 int main()
8 {
9     char s1[] = "ABCDEFGH", s2[] = "ABCDXYZ";
10
11     cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
12         << "\nmemcmp(s1, s2, 4) = " << setw( 3 )
13         << memcmp( s1, s2, 4 ) << "\nmemcmp(s1, s2, 7) = "
14         << setw( 3 ) << memcmp( s1, s2, 7 )
15         << "\nmemcmp(s2, s1, 7) = " << setw( 3 )
16         << memcmp( s2, s1, 7 ) << endl;
17     return 0;
18 }

```

**输出结果:**

```

s1 = ABCDEFG
s2 = ABCDXYZ

```

```

memcmp(s1, s2, 4) = 0
memcmp(s1, s2, 7) = -19
memcmp(s1, s2, 7) = 19

```

图 16.37 函数 memcmp

函数 memchr 在某个对象的指定个数的字节中查找首次出现某个字节（该字节用 unsigned char 类型表示）的位置。如果找到该字节，返回指向它的指针，否则返回 NULL 指针。图 16.38 中的程序在字符串 "This is a string" 中查找字符（字节）'r'。

```

1 // Fig. 16.38: fig16_38.cpp
2 // Using memchr
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *s = "This is a string";
9
10     cout << "The remainder of s after character 'r' "
11         << "is found is \"" << (char *) memchr( s, 'r', 16 )
12         << "\" " << endl;
13     return 0;
14 }

```

**输出结果:**

```
The remainder of s after character 'r' is found is "ring"
```

图 16.38 函数 memchr 的用法

函数 memset 把第二个参数中的字节复制到第一个指针参数所指向对象的指定个数的字节中。图 16.39 中的程序用函数 memst 把 'b' 复制到 string1 的前 7 个字节中。

```

1 // Fig. 16.39: fig16_39.cpp
2 // Using memset

```

```

3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char string1[ 15 ] = "BBBBBBBBBBBBBBB";
9
10    cout << "string1 = " << string1 << endl;
11    cout << "string1 after memset = "
12        << (char *) memset( string1, 'b', 7 ) << endl;
13    return 0;
14 }

```

**输出结果:**

```

string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbBBBBBBB

```

图 16.39 函数 memset 的用法

## 16.13 字符串处理库中的其他函数

字符串处理库中还有函数 strerror。图 16.40 描述了 strerror 函数。

函数原型	函数描述
char * strerror ( int errnum )	建立与 errnum 匹配的完整的文本字符串（与系统有关），返回指向该字符串的指针

图 16.40 字符串处理库中的其他字符串操作函数

函数 strerror 用错误编号作为参数，建立错误消息字符串，返回指向该字符串的指针。图 16.41 中的程序演示了函数 strerror 的用法。

```

1 // Fig. 16.41: fig16_41.cpp
2 // Using strerror
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     cout << strerror( 2 ) << endl;
9     return 0;
10 }

```

**输出结果:**

```
No such file or directory
```

图 16.41 函数 strerror 的用法

### 可移植性提示 16.7

strerror 产生的错误消息与系统有关。

## 小结

- 结构是用同一个名字引用的相关变量的集合（有时称为“聚合体”）。
- 结构中可以包含各种数据类型的变量。
- 每一个结构的定义都是以关键字 `struct` 开始的。结构定义的花括号内是结构成员的声明。
- 同一个结构内的成员必须具有惟一的名称。
- 结构的定义建立了一种用来声明变量的新的数据类型。
- 可以用初始化值列表初始化一个结构，方法是：在声明结构变量时，在变量名后用等号连接括在花括号中的初始化值列表，初始化值用逗号分开。如果初始化值的个数少于结构中的成员数目，剩余的成员自动初始化为 0（如果成员是指针，初始化为 `NULL`）。
- 可以把整个结构变量赋给同种类型的结构变量。
- 可以用同一种类型的结构变量初始化一个结构变量。
- 结构和结构的单个成员是以按值调用方式传递给函数的。而数组成员是以按引用调用方式传递给函数的。
- 传递结构变量的地址能够以按引用调用方式传递结构。
- 结构数组是自动按引用调用方式传递的。
- 要以按值调用方式传递数组，建立一个以数组作为成员的结构，然后传递这个结构。
- `typedef` 用来建立的新的类型名，而不是建立一种新的类型。它所建立的名字是以前已经定义好的类型的别名。
- 按位与运算符（`&`）需要两个操作数。如果两个操作数的相应位为 1，结构中的相应位也为 1。
- “屏蔽字”用来隐藏某些位。
- 按位或运算符（`|`）要有两个操作数。如果两个操作数中有一个操作数的相应位为 1，那么结果中的相应位也为 1。
- 每一种位运算符（除了一元按位取反运算符）都有一种相应的赋值运算符。
- 按位异或运算符（`^`）要有两个操作数。如果两个操作数的相应位只有一个为 1，结果中的相应位才为 1。
- 左移位运算符（`<<`）将其左操作数向左移动右操作数指定的位数。右边的空位用 0 填补。
- 右移位运算符（`>>`）把左操作数向右移动右操作数指定的位数。对无符号整数右移位导致的空位用 0 填补，对有符号整数右移位导致的空位用 0 或 1 填补（与机器有关）。
- 按位取反运算符（`~`）有一个操作数，它反转操作数的各位，从而产生操作数的反码。
- 位段能用所需的最少位数存储数据，从而节省了内存。
- 位段成员必须声明为 `int` 或 `unsigned` 类型。
- 声明位段的方法是：在 `unsigned` 或 `int` 成员名后依次使用冒号和位段的宽度。
- 位段宽度必须是一个整数常量，其值在 0 到机器存储 `int` 变量的位数之间。
- 如果指定的位段没有名字，那么该位段当作结构中的填充使用。
- 0 宽度的无名位段用来把下一个位段从新的字边界开始存放。
- 函数 `islower` 确定其参数是否是一个小写字母（`a~z`）。
- 函数 `isupper` 确定其参数是否是一个大写字母（`A~Z`）。
- 函数 `isdigit` 确定其参数是否是一个数字（`0~9`）。
- 函数 `isalpha` 确定其参数是否是一个大写字母（`A~Z`）或一个小写字母（`a~z`）。



- 函数 `isalnum` 确定其参数是否是一个大写字母 (A~Z)、一个小写字母 (a~z) 或一个数字 (0~9)。
- 函数 `isxdigit` 确定其参数是否是一个十六进制数字 (A~F、a~f、0~9)。
- 函数 `toupper` 把小写字母转换为大写字母, 返回大写字母。
- 函数 `tolower` 把大写字母转换为小写字母, 返回小写字母。
- 函数 `isspace` 确定其参数是否是如下空白字符之一: ' ' (空格)、'\f'、'\n'、'\r'、'\t' 和 '\v'。
- 函数 `isctrl` 确定其参数是否是如下控制符之一: '\t'、'\v'、'\f'、'\a'、'\b'、'\r' 和 '\n'。
- 函数 `ispunct` 确定其参数是否一个除空格、数字和字母之外的可打印字符。
- 函数 `isprint` 确定其参数是否是可打印字符 (包括空格)。
- 函数 `isgraph` 确定其参数是否是除空格之外的可打印字符。
- 函数 `atof` 把其参数 (用一系列表示浮点数的数字开头的字符串) 转换为一个 `double` 类型的值。
- 函数 `atoi` 把其参数 (用一系列表示整数的数字开头的字符串) 转换为一个 `int` 类型的值。
- 函数 `atol` 把其参数 (用一系列表示长整数的数字开头的字符串) 转换为一个 `long` 类型的值。
- 函数 `strtod` 把代表一个浮点数值字符序列转换为 `double` 类型。函数有两个参数: 一个字符串 (`char *` 类型) 和一个指向 `char *` 类型数据的指针。字符串中包含了要转换的字符序列, 字符串中已转换字符序列的剩余部分赋给指向 `char *` 类型数据的指针。
- 函数 `strtol` 把代表整数值的字符序列转换为 `long` 类型。函数有三个参数: 一个字符串 (`char *` 类型)、一个指向 `char *` 类型数据的指针和一个整数。字符串中包含了要被转换的字符序列, 字符串中被转换字符序列后剩余的部分赋给指向 `char *` 类型数据的指针。
- 函数 `stroul` 把代表某个 `unsigned long` 类型的整数值的字符序列转换为 `unsigned long` 类型。该函数有三个参数, 一个字符串 (`char *`), 一个指向 `char *` 的指针和一个整数。字符串包含要被转换的序列, 将转换后余下的字符串赋给指向 `char *` 类型数据的指针, 整数指定转换值的基数。
- 函数 `strchr` 查找第一次出现在字符串中的某个字符。如果找到该字符, `strchr` 返回指向它的指针, 否则返回 `NULL`。
- 函数 `strcspn` 计算其第一个字符串参数不含第二个字符串参数中字符的起始部分长度, 并返回该长度。
- 函数 `strpbrk` 在其第一个字符串参数中查找首次出现在第二个字符串参数中的字符。如果找到了第二个参数中的字符, 返回一个指向该字符的指针, 否则 `strpbrk` 返回 `NULL`。
- 函数 `strrchr` 查找最后一次出现在字符串中的某个字符。如果找到该字符, 返回指向它的指针, 否则 `strrchr` 返回 `NULL`。
- 函数 `strspn` 计算第一个字符串参数中只包含第二个字符串参数中字符的起始部分长度, 并返回该长度。
- 函数 `strstr` 在第一个字符串参数中查找首次出现其第二个字符串参数的位置。如果在第一个字符串中找到了第二个字符串, 返回一个指向包含在第一个参数中的该字符串的指针。
- 函数 `memcpy` 把第二个指针参数所指向的对象中的指定个数的字符复制到第一个指针参数所指向的对象中。函数用 `void` 指针接收指向任何类型对象的指针, 并在使用时把它转换为 `char` 类型的指针。函数 `memcpy` 把对象的字节当作字符操作。
- 函数 `memmove` 把指定个数的字节数从第二个指针参数所指向的对象复制到第一个指针参数所指向的对象中。复制过程就好像是先把指定个数的字节数从第二个指针参数所指向的对象复制到一个临时字符数组中, 然后再从临时数组复制到第一个参数。

- 函数 `memcmp` 把第一个参数中指定个数的字符与第二参数中的相应的字符进行比较。
- 函数 `memchr` 在某个对象的指定个数的字节中查找首次出现某个字节(该字节用 `unsigned char` 类型表示)的位置。如果找到该字节,返回指向它的指针,否则返回 `NULL` 指针。
- 函数 `memset` 把第二个参数中的字节 (`unsigned char` 类型)值复制到第一个指针参数所指向对象的指定个数的字节中。
- 函数 `strerror` 建立与错误号匹配的完整文本字符串(与系统有关),返回指向该字符串的指针。

## 术语

<code>^</code> bitwise exclusive OR operator	按位异或运算符	initialization of structures	结构的初始化
<code>~</code> one's complement operator	按位取反运算符	<code>isalnum</code>	
<code>&amp;</code> bitwise AND operator	按位与运算符	<code>isalpha</code>	
<code>&amp;=</code> bitwise AND assignment operator	按位与赋值运算符	<code>isctrl</code>	
<code>&lt;&lt;</code> left shift operator	左移位运算符	<code>isdigit</code>	
<code>&lt;&lt;=</code> left shift assignment operator	左移位赋值运算符	<code>isgraph</code>	
<code>^=</code> bitwise exclusive OR assignment operator	按位异或赋值运算符	<code>islower</code>	
<code>&gt;&gt;</code> right shift operator	右移位运算符	<code>isprint</code>	
<code>&gt;&gt;=</code> right shift assignment operator	右移位赋值运算符	<code>ispunct</code>	
<code> </code> bitwise inclusive OR operator	按位或运算符	<code>isspace</code>	
<code> =</code> bitwise inclusive OR assignment operator	按位或赋值运算符	<code>isupper</code>	
array of structures	结构数组	<code>isxdigit</code>	
ASCII	ASCII 码	left shift	左移位
<code>atof</code>		literal	直接量
<code>atoi</code>		mask	屏蔽字
<code>atol</code>		masking off bits	位屏蔽
bit field	位段	<code>memchr</code>	
bitwise operators	位运算符	<code>memcmp</code>	
character code	字符编码	<code>memcpy</code>	
character constant	字符常量	<code>memmove</code>	
character set	字符集	<code>memset</code>	
complementing	取反	one's complement	反码
control character	控制字符	padding	填充
<code>ctype.h</code>		pointer to a structure	指向结构的指针
delimiter	分隔符	printing character	打印字符
general utilities library	通用实用程序库	record	记录
hexadecimal digits	十六进制数字	right shift	右移位
		search string	查找字符串
		self-referential structure	自引用结构
		shifting	移位
		space-time trade-offs	“时间空间”冲突
		<code>stdlib.h</code>	

strchr	strtol
strespn	strtoul
strerror	struct
string 字符串	structure assignment 结构赋值
string constant 字符串常量	structure initialization 结构初始化
string conversion functions 字符串转换函数	structure type 结构类型
string literal 字符串直接量	tolower
string processing 字符串处理	toupper
string.h	typedef
strpbrk	unnamed bit field 无名位段
strchr	whitespace characters 空白字符
strspn	width of a bit field 位段宽度
strstr	word processing 字处理
strtod	zero-width bit field 0宽度位段

## 自测练习

### 16.1 填空:

- \_\_\_\_\_是用同一个名字引用的相关变量的集合。
- \_\_\_\_\_运算符在表达式中两个操作数的相应位都为1时才将结果设为1, 否则为0。
- 在结构定义中声明的变量称为\_\_\_\_\_。
- 如果两个操作数中的一个或两个的相应位为1, \_\_\_\_\_运算符把结果中的对应位设置为1, 否则设置为0。
- 关键字\_\_\_\_\_引出了结构的声明。
- 关键字\_\_\_\_\_可用来建立已定义好的数据类型的别名。
- 如果两个操作数的相应位只有一个为1, \_\_\_\_\_运算符把结果中的对应位设置为1, 否则设置为0。
- 按位与运算符&经常用于\_\_\_\_\_某些位, 用来在选择一个位串的某些位并把其他位设置为0。
- 结构的名字称为结构的\_\_\_\_\_。
- 用\_\_\_\_\_运算符和\_\_\_\_\_运算符访问结构的成员。
- \_\_\_\_\_运算符和\_\_\_\_\_运算符分别把一个值的位向左和向右移动。

### 16.2 判断下列各题是否正确。如果不正确, 请说明原因。

- 结构只能包含一种数据类型。
- 不同结构的成员必须有惟一的名字。
- 关键字 typedef 用来定义新的数据类型。
- 结构总是以按引用调用的方式传递给函数。

### 16.3 用一条或一组语句分别完成下列要求。

- 定义一个结构 Part, 它包含 int 类型的变量 partNumber 和 char 类型的数组 partName, 数组的长度为 25 个字符。
- 定义 PartPtr 是类型 part \* 的别名。

- c) 声明变量 a、数组 b[10]、指针 ptr 的类型为 Part。
- d) 从键盘读取变量 a 的两个成员。
- e) 把变量 a 的成员值赋给数组 b 第 3 个元素。
- f) 把数组 b 的地址赋给指针变量 ptr。
- g) 用变量 ptr 和结构指针运算符打印数组 b 第 3 个元素的值。

16.4 指出下列语句中的错误。

- a) 假设 struct card 包含两个 char 类型的指针 face 和 suit。变量 c 声明为 Card 类型，变量 cPtr 声明为 Card 类型的指针。c 的地址已经赋给了变量 cPtr。

```
cout << *cPtr.face << endl;
```

- b) 假设 struct Card 包含两个 char 类型的指针 face 和 suit。数组 hearts[13] 被声明为 Card 类型。下面一条语句用来打印该数组的第 11 个元素的成员 face:

```
cout << hearts.face << endl;
```

- c) struct Person {  
     char lastName[ 15 ];  
     char firstName[ 15 ];  
     int age;  
 }

- d) 假定变量 p 已经声明为 Person 类型，变量 c 声明为 Card 类型。

```
P = c;
```

16.5 用一条语句完成下列要求。假设变量 c (存储一个字符)、x、y 和 z 的类型是 int，变量 d、e 和 f 的类型是 float，变量 ptr 的类型是 char\*，数组 s1[100] 和 s2[100] 的类型是 char。

- a) 把存储在 c 中的字符转换为大写字母，结果赋给变量 c。
- b) 确定变量 c 的值是否是一个数字。在显示结果时，用条件运算符 (见图 16.17、图 16.18 和图 16.19) 打印出 "is a" 或 "is not a"。
- c) 把字符串 "1234567" 转换为 long 类型并打印出结果。
- d) 确定变量 c 是否是一个控制字符。在显示结果时，用条件运算符打印出 "is a" 或 "is not a"。
- e) 把 s2 中最后一次出现变量 c 的内存单元赋给 ptr。
- f) 把字符串 "8.63582" 转换为 double 类型并打印出值。
- g) 确定 c 的值是否是一个字母。在显示结果时，用条件运算符打印出 "is a" 或 "is not a"。
- h) 把 s1 中首次出现 s2 的内存单元赋给 ptr。
- i) 确定变量 c 的值是否是一个可打印字符。在显示结果时，用条件运算符打印出 "is a" 或 "is not a"。
- j) 把 s1 中首次出现 s2 中字符的内存单元赋给 ptr。
- k) 把 s1 中首次出现变量 c 的内存单元赋给 ptr。
- l) 把字符串 "~21" 转换为 int 类型并打印出值。

## 自测练习答案

- 16.1 a) 结构。b) 按位与 (&)。c) 成员。d) 按位或 (|)。e) struct。f) typedef。g) 按位异或 (^)。  
 h) 屏蔽。i) 标记。j) 结构成员 (.)、结构指针 (->)。k) 左移位运算符、右移位运算符。

- 16.2 a) 不正确。结构可包含许多数据类型。  
 b) 不正确。不同结构的成员可以有相同的名字,但是同一个结构的成员不能有相同的名字。  
 c) 不正确。关键字 `typedef` 用来为已定义好的数据类型定义新的名字(别名)。  
 d) 不正确。结构总是按值调用传递的。
- 16.3 a) 

```
struct Part {
    int partNumber;
    char partName[ 26 ];
};
```

  
 b) 

```
typedef Part * PartPtr;
```

  
 c) 

```
Part a, b[ 10 ], *ptr;
```

  
 d) 

```
cin >> a.partNumber >> a.partName;
```

  
 e) 

```
b[ 3 ] = a;
```

  
 f) 

```
ptr = b;
```

  
 g) 

```
cout << ( ptr + 3 ) ->partNumber << ' '
      << ( ptr + 3 ) ->partName << endl;
```
- 16.4 a) 错误: 应该用括号把 \* cPtr 括起来, 否则表达式的计算顺序不正确。  
 b) 错误: 数组没有下标。表达式应该为 `hearts[ 10 ].face`。  
 c) 错误: 结构定义的最后要有分号。  
 d) 错误: 不同结构类型的变量不能相互赋值。
- 16.5 a) 

```
c = toupper( c );
```

  
 b) 

```
cout << '\\' << c << '\\' "
      << ( isdigit( c ) ? "is a" : "is not a" )
      << " digit" << endl;
```

  
 c) 

```
cout << atol( "1234567" ) << endl;
```

  
 d) 

```
cout << '\\' << c << '\\' "
      << ( iscntrl( c ) ? "is a" : "is not a" )
      << " control character" << endl;
```

  
 e) 

```
ptr = strrchr( s1, c );
```

  
 f) 

```
cout << atof( "8.63582" ) << endl;
```

  
 g) 

```
cout << '\\' << c << '\\' "
      << ( isprint( c ) ? "is a" : "is not a" )
      << " letter" << endl;
```

  
 h) 

```
ptr = strrchr( s1, s2 );
```

  
 i) 

```
cout << '\\' << c << '\\' "
      << ( isprint( c ) ? "is a" : "is not a" )
      << " printing character " << endl;
```

  
 j) 

```
ptr = strpbrk( s1, s2 );
```

  
 k) 

```
ptr = strchr( s1, c );
```

  
 l) 

```
cout << atoi( "-21" ) << endl;
```

## 练习

- 16.6 给出下列结构和联合体的定义。
- a) 结构 `Inventory`, 包含字符数组 `partName[ 10 ]`、整数变量 `partNumber`、浮点数变量 `price`、整数变量 `stock` 和整数变量 `reorder`。
- b) 结构 `Address`, 包含字符数组 `streetAddress[ 25 ]`、`city[ 20 ]`、`state[ 3 ]` 和 `zipCode[ 6 ]`。
- c) 结构 `Student`, 包含字符数组 `firstName[ 15 ]`、`lastName[ 15 ]` 以及类型为 b) 中的 `struct`

- Address 的变量 homeAddress。
- d) 结构 Test, 包含 16 个宽度为 1 位的位段, 位段名从 a 到 p。
- 16.7 根据给出的结构定义和变量声明, 写出访问下面结构成员的表达式。

```
struct Customer {
    char lastName[ 15];
    char firstName[ 15];
    int customerNumber;

    struct {
        char phoneNumber[ 11];
        char address[ 50];
        char city[ 15];
        char state[ 3];
        char zipCode[ 6];
    } personal;
} customerRecord, *customerPtr;

customerPtr = &customerRecord;
```

- a) 结构 customerRecord 中的成员 lastName。
  - b) customerPtr 所指向的结构中的成员 lastName。
  - c) 结构 customerRecord 中的成员 firstName。
  - d) customerPtr 所指向的结构中的成员 firstName。
  - e) 结构 customerRecord 中的成员 customerNumber。
  - f) customerPtr 所指向的结构中的成员 customerNumber。
  - g) 结构 customerRecord 中的 personal 的成员 phoneNumber。
  - h) customerPtr 所指向的结构中的 personal 的成员 phoneNumber。
  - i) 结构 customerRecord 中的 personal 的成员 address。
  - j) customerPtr 所指向的结构中的 personal 的成员 address。
  - k) 结构 customerRecord 中的 personal 的成员 city。
  - l) customerPtr 所指向的结构中的 personal 的成员 city。
  - m) 结构 customerRecord 中的 personal 的成员 state。
  - n) customerPtr 所指向的结构中的 personal 的成员 state。
  - o) 结构 customerRecord 中的 personal 的成员 zipCode。
  - p) customerPtr 所指向的结构中的 personal 的成员 zipCode。
- 16.8 用图 16.2 中的高效洗牌算法修改图 16.14 中的程序。以图 16.3 的两列格式打印出结果, 在每张牌前标出其颜色。
- 16.9 编写一个程序, 把一个整数变量右移 4 位, 分别将移位之前和之后的整数按位打印出来。读者的系统在空位上是补 0 还是补 1?
- 16.10 如果读者的计算机使用 4 字节的整数, 修改图 16.5 中的程序, 使它操作 4 字节的整数。
- 16.11 把一个无符号整数左移 1 位等价于把它乘以 2。编写一个函数 power2, 它有两个整数参数 number 和 pow, 并计算:

$$\text{number} * 2^{\text{pow}}$$

用左移位运算符计算这个结果, 再分别以整数形式和位形式打印出结果。

- 16.12 左移位运算可用来把两个字符值合并在一个两字节的无符号整数变量中。编写一个程序，从键盘读取两个字符，把它们传递给函数 `packCharacters`。为了将这两个字符合并在一个无符号整数变量中，先把第一个字符赋给无符号变量，把变量左移位8位，然后用按位或运算符把该变量和第二个字符组合在一起。为证实这两个字符合并在了无符号整数变量中，以位格式分别打印出这两个字符在合并前和合并后的值。
- 16.13 用右移位运算符、按位与运算符和屏蔽字编写函数 `unpackCharacters`，参数是练习 16.12 的无符号整数。函数把该无符号整数分解成两个字符。分解方法是：把这个无符号整数和屏蔽字 65280 (11111111 00000000) 结合，将结果右移8位，再把所得的值赋给 `char` 类型的变量。然后把原来的无符号整数和屏蔽字 255 (00000000 11111111) 结合。把所得的结果赋给另一个 `char` 类型的变量。为了证实分解的正确性，先把分解前的无符号整数按位形式打印出来，然后再把分解后的两个字符按位的形式打印出来。
- 16.14 如果读者系统使用的整数占4个字节，修改练习 16.12 中的程序，使它对4个字符合并。
- 16.15 如果读者系统使用的整数占4个字节，修改练习 16.13 中的函数 `unpackCharacters`，使它能够分解所包含的4个字符。要建立对4个字符分解的屏蔽字，把255左移位0位、8位、16位或32位（即8的倍数，取决于要分解的是哪一个字节）。
- 16.16 编写一个程序，把一个无符号整数的各个字节反序排列。程序从用户那里读取一个无符号整数，用函数 `reverseBits` 反序打印出该整数的各位。为证实反序排列的正确性，分别打印出整数在反序排列前后的各位。
- 16.17 修改图 16.5 中的函数 `displayBits`，使它能够在使用2字节整数和使用4字节整数的两种计算机上移位。提示：用 `sizeof` 运算符确定在特定机器上整数所占用的字节数。
- 16.18 编写一个程序，从键盘输入字符，用字符处理库中的每个函数测试这个字符并打印每个函数返回值。
- 16.19 如下所示的程序用函数 `multiple` 判断从键盘输入的整数是否是某个整数 `x` 的倍数。研究该函数，判断 `x` 的值。

```
1 // This program determines if a value is a multiple of x
2 #include <iostream.h>
3
4 int multiple( int );
5
6 int main()
7 {
8     int y;
9
10    cout << "Enter an integer between 1 and 32000:";
11    cin >> y;
12
13    if ( multiple( y ) )
14        cout << y << " is a multiple of x" << endl;
15    else
16        cout << y << "is not a multiple of x" << endl;
17
18    return 0;
19 }
20
21 int multiple( int num )
22 {
```

```

23     int mask = 1, mult = 1;
24
25     for ( int i = 0; i < 10; i++, mask <= 1 )
26         if ( ( num & mask ) != 0 ) {
27             mult = 0;
28             break;
29         }
30
31     return mult;
32 }

```

16.20 下面的程序有什么作用？

```

1  #include <iostream.h>
2
3  int mystery( unsigned );
4
5  int main()
6  {
7      unsigned x;
8
9      cout << "Enter an integer: ";
10     cin >> x;
11     cout << "The result is" << mystery( x ) << endl;
12     return 0;
13 }
14
15 int mystery( unsigned bits )
16 {
17     unsigned mask = 1 << 15, total = 0;
18
19     for ( int i = 0; i < 16, i++, bits <= 1;
20         if ( ( bits & mask ) == mask )
21             ++ total;
22
23     return total % 2 == 0 ? 1 : 0;
24 }

```

- 16.21 编写一个程序，用 `istream` 的成员函数 `getline`（见第 11 章）把一行文本输入到字符数组 `s[100]` 中。再分别以大写字母和小写字母输出这行文本。
- 16.22 编写一个程序，输入 4 个代表整数值的字符串，把该字符串转换为整数，对整数值求和，并打印这 4 个值的和。
- 16.23 编写一个程序，输入 4 个表示浮点数的字符串，把字符串转换为 `double` 类型的值，求出并打印这 4 个值的和。
- 16.24 编写一个程序，从键盘读取一行文本和一个查找串。用函数 `strstr` 查找文本中第一次出现查找串的位置，把这个位置赋给 `char *` 类型的变量 `searchPtr`。如果找到查找串，打印出该行文本中查找串之后的部分，然后再用 `strstr` 查找在文本中下一次出现查找串的位置，如果第二次找到，再打印出该行文本中查找串之后的部分。提示：第二次调用函数 `strstr` 时应该把 `searchPtr+1` 作为它的第一个参数。
- 16.25 在练习 16.24 中的基础上编写一个程序，读取几个文本行和一个查找串，用函数 `strstr` 确定查找串在文本中出现的次数，然后打印出结果。



- 16.26 编写一个程序,读取几个文本行和一个检索字符,用函数 `strchr` 确定这个字符在所有文本行中出现的次数。
- 16.27 在练习 16.26 的基础上编写一个程序,读取几个文本行,用函数 `strchr` 确定字母表中的每个字母在所有文本行中的出现次数(不区分大小写)。把每个字母出现的次数存放在一个数组中,然后以表格的形式打印出最后的结果。
- 16.28 附录 B 中列出了 ASCII 字符集中所有字符的数字代码。根据这个表判断下列说法是否正确。
- a) 字母“A”排在字母“B”之前。
  - b) 数字“9”排在数字“0”之前。
  - c) 常用到的加、减、乘、除符号排在所有数字之前。
  - d) 数字在字母之前。
  - e) 按升序排序字符串的排序程序会把右花括号排在左花括号之前。
- 16.29 编写一个程序,读取一组字符串,只把以字母“b”开头的那些字符串打印出来。
- 16.30 编写一个程序,读取一组字符串,只把以字母“ED”结尾的那些字符串打印出来。
- 16.31 编写一个程序,读取一个 ASCII 码,打印出与之对应的字符。修改这个程序,使它产生范围在 000 到 255 之间所有可能的三位 ASCII 码,打印出与之对应的字符。运行这个程序会发生什么现象?
- 16.32 根据附录 B 中的 ASCII 字符表,编写图 16.16 中的字符处理函数。
- 16.33 编写图 16.20 中把字符串转换成数字的函数。
- 16.34 编写图 16.27 中的字符串查找函数。
- 16.35 编写图 16.34 中的内存块操作函数。
- 16.36 (项目: 拼写检查程序)许多常用字处理软件包都内置了拼写检查程序。我们用 Microsoft Word 5.0 的拼写检查程序编写本书,发现不论我们如何认真编写,Word 总能找出一些我们无法用手工方法找到的拼写错误。

在这个项目中,要开发自己的拼写检查程序。我们会提供一些提示,然后读者要增加更多功能。你会发现,用计算机化字典作为单词源非常方便。

为什么输入时会有拼写错误呢?有时是因为我们记不住单词的拼法,有时则是两个字母颠倒了(例如“default”而不是“default”),有时是一个字母重输了(如“hanndy”而不是“handy”),有时是按错了键(如“biryhday”而不是“birthday”),等等。

设计和实现 C++ 拼写检查程序。程序维护字符串 `wordList` 数组。你可以输入这些字符串,或者用计算机化字典作为单词源。

程序请求用户输入一个单词,然后对照 `wordList` 数组中的单词。如果该单词在数组中存在,则程序打印“Word is spelled correctly”。

如果该单词在数组中不存在,则程序打印“word is not spelled correctly”。然后程序查找 `wordList` 中的其他项目,猜测用户要输入的单词。例如,可以通过各种相邻字母的置换发现 `wordList` 中的“default”项目。当然,程序还要检查其他置换,如“dfeault”、“deafult”、“defalut”和“defaultl”。找到匹配 `wordList` 项目的新词时,打印如下消息,“Did you mean “default?””。

实现其他测试,如把重叠字母换为单个字母,或用其他方法增加拼写检查程序的功能。

# 第17章 预处理器

## 教学目标

- 能够用 `#include` 开发大型程序
- 能够用 `#define` 建立宏和带有参数的宏
- 了解条件编译
- 能够在条件编译时显示错误消息
- 能够用 `assert` 测试表达式的值是否正确

## 17.1 简介

本章介绍预处理器。预处理发生在编译之前，包括把其他文件包含到要编译的文件中、定义符号常量和宏、程序代码的条件编译以及预处理指令的条件执行。所有的预处理指令都是用#开头的。一行预处理指令的前面只能出现空白字符。预处理指令不是C++语句，因此不用分号(;)结尾。预处理指令要在编译之前处理完毕。

### 常见编程错误 17.1

将分号放在预处理指令末尾可能产生各种错误，这取决于预处理指令的类型。

### 软件工程视点 17.1

许多预处理特性（特别是宏）更适合C语言编程而不是C++编程。C++程序员熟悉预处理指令有助于处理C语言遗留代码。

## 17.2 预处理指令 `#include`

预处理指令 `#include` 用指定文件的一份副本取代这条预处理指令。`#include` 指令有如下两种形式：

```
#include <filename>
#include "filename"
```

这两种形式的差别在于预处理器查找要包含的文件的路径不同。如果用双引号标识文件名，预处理器在正在编译的程序所在的目录中查找要包含的文件，该方法通常用来把程序员定义的头文件包含到程序中。如果用尖括号标识文件名（用来查找标准库头文件），预处理器就用与实现相关的方式查找要包含的文件，通常是在预先设计的目录中查找。如果文件名放在引号中，则预处理器首先在所编译文件的同一目录中查找，然后使用与查找三角括号中的文件名相同的实现相关方式进行查找。这种方法通常用于包括程序员自定义的头文件。

`#include`指令通常用来把标准库头文件(如头文件`iostream.h`和`iomanip.h`)包含到程序中。`#include`指令还和要一起编译的多个源文件组成的程序一起使用。程序员经常建立一个头文件,该头文件包含了单个程序文件的定义和声明。这些声明和定义的例子有声明结构、联合体、枚举以及函数原型等等。

## 17.3 预处理指令 `#define`: 符号常量

预处理指令`#define`用来建立符号常量(用符号表示的常量)和宏(用符号定义的操作)。`#define`指令的格式为:

```
#define 标识符 替换文本
```

当文件中出现这一行时,以后出现的所有该标识符都会在程序编译前自动用替换文本取代。例如:

```
#define PI 3.14159
```

用数值常量3.14159取代了以后出现的所有符号常量PI。程序员可以用符号常量为某个常量建立一个名字,然后在整个程序中使用这个名字。如果需要修改整个程序中用到的该常量,可以在`#define`指令中作一次性的修改,然后再重新编译程序就会自动修改出现在程序中的所有常量。注意:预处理指令`#define`之后出现的符号常量是用符号常量名右边的所有的文本替换的。例如,预处理指令`#define PI = 3.14159`使预处理器用`= 3.14159`取代其后出现的所有的PI,这会产生许多微妙的逻辑错误和语法错误。用新值重定义符号常量也是一个错误。注意C++中的`const`变量优于符号常量。常量变量具有特定数据类型,调试器能通过名称访问该常量变量。用替换文本替换常量符号之后,调试器只能访问替换文本。`const`变量的缺点是需要数据类型长度的内存位置,而符号常量不需要增加内存。

### 常见编程错误 17.2

在定义符号常量的文件之外使用该符号常量是个语法错误。

### 编程技巧 17.1

给符号常量取有意义的名字可提高程序的可读性。

## 17.4 预处理指令 `#define`: 宏

(说明: C++程序员熟悉预处理指令有助于处理C语言遗留代码。在C++中,宏已经换成模板和内联函数)。宏是在预处理指令`#define`中定义的一种操作。和符号常量一样,程序中的宏标识符也在程序编译之前被替换文本取代。可以定义带有或不带有参数的宏。预处理器就像处理符号常量一样处理不带参数的宏。对带有参数的宏的处理方式是:先用替换文本取代参数,然后再把宏展开,即用替换文本取代程序中的标识符和参数表。注意:宏参数没有数据类型检查,宏只用于文本替换。

考虑如下所示的带有一个求圆面积参数的宏定义:

```
#define CIRCLE_AREA (x) ( PI * (x) * (x) )
```

不论文件中何时出现 `CIRCLE_AREA(x)`, 替换文本中的 `x` 都用 `x` 的值取代, 符号常量 `PI` 用前面定义的值取代, 然后展开程序中的宏。例如, 下列语句:

```
area = CIRCLE_AREA (4);
```

展开成为:

```
area = ( 3.14159 * (4) * (4) );
```

因为该表达式只由常量组成, 所以在编译时就能计算出该表达式的值并赋给变量 `area`。用括号把替换文本中的 `x` 括起来是为了在宏参数是表达式时能够强制编译器以正确的顺序计算表达式的值。例如, 下列语句:

```
area = CIRCLE_AREA (c + 2);
```

展开成为:

```
area = ( 3.14159 * (c + 2) * (c + 2) );
```

表达式中的括号使得表达式能够以正确的顺序计算。如果去掉括号, 宏就会被展开为:

```
area = 3.14159 * c + 2*c + 2;
```

就会错误地求值成下列表达式:

```
area = ( 3.14159 * c) + (2 * c) +2;
```

#### 常见编程错误 17.3

忘记替换文本中的用于宏参数的括号。

可以把宏 `CIRCLE_AREA` 定义为一个函数。函数 `circleArea`:

```
double circleArea ( double x ) { return 3.14159 * x * x; }
```

与宏 `CIRCLE_AREA` 完成同样的计算, 但是函数 `circleArea` 需要函数调用的开销。使用宏 `CIRCLE_AREA` 的优点是宏直接把代码插入到程序中(避免了函数调用的开销), 并且保持了程序的可读性(因为单独定义了 `CIRCLE_AREA` 的计算, 并且 `CIRCLE_AREA` 中的命名也是有意义的), 不足之处是计算了两次参数。另外, 每次程序中出现宏时, 都要展开宏。如果宏很大, 则程序长度会变得很大。这样, 执行速度与程序长度就形成一对矛盾(磁盘空间可能不多)。注意内联函数(见第 3 章)既能实现宏的性能又能实现函数的软件工程优势。

#### 性能提示 17.1

有时通过宏替换用内联代码实现的函数调用, 这省去了函数调用的开销。内联函数优于宏, 因为其提供函数的类型检查服务。

如下的宏定义带有两个用来计算矩形面积的参数:

```
# define RECTANGLE_AREA (x,y) ( (x) * (y) )
```

不论程序中何时出现 `RECTANGLE_AREA(x, y)`, 替换文本中的 `x` 和 `y` 都被定义中的 `x` 和 `y` 值取代, 并且用展开后的宏取代宏名。例如, 下列语句:

```
rectArea = RECTANGLE_AREA (a+4, b+7);
```

展开成为:

```
rectArea = { (a+4)* (b+7) };
```

计算出的表达式的值赋给变量 rectArea。

宏或符号常量的替换文本通常在同一行中( #define 指令中的标识符之后的所有文本)。如果替换文本超过一行, 必须在该行的最后加上反斜杠(\), 反斜杠表示替换文本继续到下一行。

可以用预处理指令 #undef 结束符号常量和宏的作用。指令 #undef 取消符号常量和宏名的定义。符号常量或宏名的范围从其定义开始到用 #undef 取消其定义或到文件结束为止。取消了符号常量和宏名的定义后, 可以用 #define 指令重新定义它。

标准库中的函数有时基于其他库函数定义为一个宏。在头文件 <stdio.h> 中定义的一个常见的宏是:

```
#define getchar ()getc (stdin)
```

getchar 的宏定义用函数 getc 从标准输入流中读取一个字符。头文件 <stdio.h> 中的函数 putchar 以及头文件 <ctype.h> 中的字符处理函数也经常用宏实现。注意: 有副作用(即修改变量的值)的表达式不应该传递给宏, 因为可能会多次计算宏的参数。

## 17.5 条件编译

条件编译能够让程序控制预处理指令的执行和程序代码的编译。每一个条件预处理指令计算一个常量整数表达式的值, 以决定代码是否编译。不能在预处理指令中计算强制类型转换表达式、sizeof 表达式和枚举常量。

条件预处理指令的结构与 if 选择结构非常类似。考虑如下所示的预处理代码:

```
#if ! defined(NULL)
    #define NULL 0
#endif
```

这些预处理指令确定是否定义了符号常量 NULL。如果定义了 NULL, 表达式 defined(NULL) 的计算结果为 1, 否则为 0。如果计算结果为 0, 那么 ! defined(NULL) 的计算结果为 1, 从而把 NULL 定义为 0, 否则就跳过 #define 指令。每一个 #if 结构都是用 #endif 结束的。可以把 #if defined (name) 和 #if ! defined (name) 缩写为 #ifdef 和 #ifndef。可以用 #elif (等价于 if 结构中的 else if) 和 #else (等价于 if 结构中的 else) 指令测试包含多个部分的条件预处理结构。

在程序开发过程中, 程序员经常发现需要将一大段代码变成注释, 从而防止编译器编译这段代码。如果代码中包含了注释语句, 那么 /\* 和 \*/ 是不能用来完成这个任务的。这时, 程序员可以使用如下的预处理结构:

```
#if 0
    不编译的代码
#endif
```

要让编译器编译这段代码, 可以把原来的 0 改为 1。

条件编译通常用来帮助调试程序。通过输出语句打印变量的值来确定控制流的正确性。这些输出的语句可放在条件预处理指令之间, 因此直到调试操作完成后才编译这些语句。例如:

```
#ifdef DEBUG
    cerr( "Variable x = "<< x << endl;
#endif
```

如果在 `#ifdef DEBUG` 之前定义符号常量 `DEBUG` (即指令 `#define DEBUG`)，就编译 `cerr` 语句。在完成调试后，从程序中去掉 `#define` 指令，那么就会在编译过程中忽略为调试而插入的输出语句。在大型程序中，可能要定义多个符号常量，并用这些符号常量控制源文件某个部分的条件编译。

#### 常见编程错误 17.4

把用于调试目的的多条条件编译输出语句放在 C++ 只要求一条语句的地方会造成语法和逻辑错误。这种情况下，条件编译语句应该使用复合语句。当编译该程序时，程序的控制流没有变化。

## 17.6 预处理指令 `#error` 和 `#pragma`

### 预处理指令 `#error`

#### `#error` 标记

打印一个与程序实现过程有关的消息，消息中包含了在指令中指定的标记。标记是用空格分开的字符序列。例如：

```
#error 1 - Out of range error
```

包含了 6 个标记。在常见的 C++ 编译器中，当预处理器处理 `#error` 指令时，指令中的标记就会当作错误消息显示出来，终止处理并且不编译这个程序。

#### 预处理指令 `#pragma`：

#### `#pragma` 标记

引起实现所定义的动作，忽略不能被实现所识别的程序。例如，C++ 编译器可能识别多个程序，从而使程序员充分利用 C++ 编译器的特定功能。有关 `#error` 和 `#pragma` 的更详细的内容，参见各自的 C++ 手册。

## 17.7 运算符 `#` 和 `##`

预处理器的 `#` 和 `##` 运算符用于 C++ 和 ANSI C。运算符 `#` 把替换文本的标记转换为带引号的字符串。考虑如下所示的宏定义：

```
#define HELLO( x ) cout << "Hello," #x << endl
```

当 `HELLO(John)` 出现在程序文件中时，展开成为：

```
cout << "Hello," "John" << endl
```

字符串 `"John"` 取代了替换文本中的 `#x`。因为用空格分开的字符串会在处理过程中被连接在一起，因此上述的语句等价于下列语句：

```
cout << "Hello, John" << endl
```

注意，因为运算符#的操作数引用了宏的参数，所以#必须与带有参数的宏一起使用。

运算符##把两个标记连在一起。考虑如下所示的宏定义：

```
#define TOKENCONCAT( x, y ) x ## y
```

当TOKENCONCAT出现在程序中时，将其参数连在一起并用来替换这个宏。例如，程序中的TOKENCONCAT(O, K)被OK替换。运算符##必须有两个操作数。

## 17.8 行号

预处理指令#line使得其后的源代码行从指定的符号常量整数值开始重新编号。指令：

```
#line 100
```

使得下一条源代码行的行号从100开始。#line指令中可包含某个文件名。指令：

```
#line 100 "file1.c"
```

表明下一条源代码行的行号从100开始，并且用于编译消息的文件名为"file1.c"。这种指令通常用来帮助生成更有意义的语法错误消息和警告消息。源文件中并不出现行号。

## 17.9 预定义的符号常量

预定义的符号常量有5种（见图17.1）。每一个预定义的符号常量标识符都是用两个下划线开始和结束。这些标识符以及17.5节所用到的defined的标识符不能用在#define和#undef指令中。

符号常量	解释
__LINE__	当前源代码行的行号（某个整数常量）
__FILE__	假定的源文件名（某个字符串）
__DATE__	编译源文件的日期（"Mmm dd yyyy"形式的字符串，如"Jan 19 1991"）
__TIME__	编译源文件的时间（具有"hh: mm: ss"形式的字符串直接量）
__STDC__	整数常量1，表示实现了兼容的ANSI

图 17.1 预定义的符号常量

## 17.10 断言（宏assert）

在头文件assert.h中定义的宏assert用来测试表达式的值。如果表达式的值为0，那么assert就打印出错消息，并调用通用实用程序库stdlib.h中的函数abort终止程序的执行。这是一个有用的调试工具，可以测试一个变量是否具有某个正确的值。例如，假定程序中的变量不应该大于10，那么可以用断言测试x的值，并在x的值不正确时打印出错消息。所用的语句如下所示：

```
assert ( x <= 10 );
```

如果x大于10，那么就会打印出包含行号和文件名的错误消息并终止程序的执行，然后程序员可以把查找错误的重点放在该代码区。如果定义了符号常量NDEBUG，其后的断言将被忽略。因此，如果不再需要断言（当调试完成后），那么可把代码行：

```
# define NDEBUG
```

插入到程序文件中，而无需手工删除 `assert`。

如今大部分的C++编译器都包括异常处理，C++程序员更愿意使用异常而不是断言。但是在程序员处理C遗留代码时，断言还是很有价值的。

## 小结

- 所有的预处理指令都是以#开头。
- 一行预处理指令的前面只能出现空白字符。
- 预处理指令 `#include` 把指定文件的一份副本包含到程序中。如果文件名用双引号括起来，那么预处理器在所编译的文件的同一目录中查找要被包含的文件。如果用尖括号(<和>)把文件名括起来，那么预处理器就以实现时定义方式查找要被包含的文件。
- 预处理指令 `#define` 用来建立符号常量和宏。
- 符号常量是常量的名字。
- 宏是用预处理指令 `#define` 定义的操作。可以定义带有和不带有参数的宏。
- 宏和符号常量的替换文本是同一行中 `#define` 指令之后的所有的文本。如果宏或符号常量的替换文本超过一行，那么要在该行的最后加上反斜杠(\)，反斜杠表示替换文本继续到下一行。
- 可以用预处理指令 `#undef` 终止符号常量和宏的作用，它取消符号常量或宏的定义。
- 符号常量和宏的范围从其定义开始到用 `#undef` 取消其定义或到文件结束为止。
- 条件编译使程序员能够控制预处理指令的执行和程序代码的编译。
- 条件预处理指令计算常量整数表达式的值，不能在预处理指令中计算强制类型转换表达式、`sizeof` 表达式和枚举常量的值。
- 每一个 `#if` 结构都用 `#endif` 结束。
- 指令 `#ifdef` 和 `#ifndef` 分别是 `#if defined(name)` 和 `#if !defined(name)` 的缩写。
- 可以用 `#elif` (类似于 `if` 结构中的 `else if`) 和 `#else` (等价于 `if` 结构中的 `else`) 测试包含多个部分的条件预处理指令。
- `#error` 指令打印与实现有关的消息，消息中包含了在指令中指定的标记。
- 指令 `#program` 引起实现所定义的动作。如果实现不识别该程序，就忽略该程序。
- 运算符 `#` 把替换文本的标记转换为用双引号括起来的字符串。因为 `#` 的操作数必须是宏的参数，所以它必须用于带参数的宏。
- 运算符 `##` 连接两个标记，它必须有两个操作数。
- 预处理指令 `#line` 使源代码行从指定的常量整数值开始重编行号。
- 预定义的符号常量有5种。常量 `__LINE__` 是当前源代码行的行号(某个整数常量)，常量 `__FILE__` 是假定的源文件名(某个字符串)，常量 `__DATE__` 是编译源文件的日期(字符串)，常量 `__TIME__` 是编译源文件的时间(字符串)，常量 `__STDC__` 为1，表示其实现与ANSI兼容。注意，每一个预定义的符号常量都是用两个下划线开始和结束。
- 在头文件 `assert.h` 中定义的宏 `assert` 用来测试表达式的值。如果表达式的值为0，那么 `assert` 就打印错误消息，并调用函数 `abort` 终止程序的执行。



## 术语

#define	concatenation preprocessor operator## 预处理器的连接运算符 ##
#elif	
#else	conditional compilation 条件编译器
#endif	conditional execution of preprocessor directives 预处理指令的条件执行
#error	
#if	convert-to-string preprocessor 转换成字符串的预处理器
#ifdef	
#ifndef	operator# 运算符 #
#include <filename>	debugger 调试器
#include "filename"	expand a macro 宏的展开
#line	header file 头文件
#pragma	macro 宏
#undef	macro with arguments 带有参数的宏
\ ( backslash ) continuation character 续行符	predefined symbolic constants 预定义的符号常量
\ ( 反斜杠 )	preprocessing directive 预处理指令
__DATE__	preprocessor 预处理
__FILE__	replacement text 替换文本
__LINE__	scope of a symbolic constant or macro 符号常量或宏的范围
__STDC__	standard library header files 标准库头文件
__TIME__	stdio.h
abort	stdlib.h
argument 参数	symbolic constant 符号常量
assert	
assert.h	

## 自测练习

## 17.1 填空:

- 每一条预处理指令都必须用 \_\_\_\_\_ 开头。
- 可以用 \_\_\_\_\_ 和 \_\_\_\_\_ 指令扩展条件编译结构来测试多种条件。
- \_\_\_\_\_ 指令建立宏和符号常量。
- 同一行的预处理指令前只能出现 \_\_\_\_\_ 字符。
- \_\_\_\_\_ 指令取消符号常量和宏名的作用。
- \_\_\_\_\_ 和 \_\_\_\_\_ 指令是 #if defined (name) 和 #if ! defined (name) 的缩写。
- \_\_\_\_\_ 使程序员能够控制预处理指令的执行和程序代码的编译。
- 如果宏 \_\_\_\_\_ 计算出的表达式的值等于 0, 那么它就打印出一条消息并终止程序的执行。
- \_\_\_\_\_ 指令把文件插入到另一个文件中。
- 运算符 \_\_\_\_\_ 连接它的两个参数。

- k) 运算符 \_\_\_\_\_ 把其操作数转换为一个字符串。  
 l) 字符 \_\_\_\_\_ 表示符号常量或宏的替换文本继续到下一行。  
 m) 预处理指令 \_\_\_\_\_ 使源代码行从指定的常量整数值开始重编行号。
- 17.2 编写一个程序，打印出图 17.1 中列出的预定义的符号常量值。
- 17.3 编写出完成如下要求的预处理指令。
- 定义符号常量 YES 的值为 1。
  - 定义符号常量 NO 的值为 0。
  - 包含头文件 common.h，该文件和被编译的文件在同一个目录中。
  - 对文件中剩下的行从行号 3000 开始编号。
  - 如果定义了符号常量 TRUE，则取消其定义并重定义为 1。不使用 #ifdef。
  - 如果定义了符号常量 TRUE，则取消其定义并重定义为 1。使用 #ifdef。
  - 如果符号常 ACTIVE 不等于 0，定义符号常量 INACTIVE 为 0，否则定义 INACTIVE 为 1。
  - 定义计算正方体体积的带有一个参数的宏 CUBE\_VOLUME。

### 自测练习答案

- 17.1 a) #。 b) #elif、#else。 c) #define。 d) 空白。 e) #undef。 f) #ifdef、#ifndef。 g) 条件编译。 h) assert。 i) #include。 j) ##。 k) #。 l) \。 m) #line。

17.2 #include <iostream.h>

```
main()
{
    cout << " __LINE__ =" << __LINE__ << endl;
    cout << " __FILE__ =" << __FILE__ << endl;
    cout << " __DATE__ =" << __DATE__ << endl;
    cout << " __TIME__ =" << __TIME__ << endl;
    cout << " __STDC__ =" << __STDC__ << endl;
}
```

输出结果：

```
__LINE__ = 5
__FILE__ = macros.c
__DATE__ = Nov 10 1997
__TIME__ = 10:23:47
__STDC__ = 1
```

- 17.3 a) #define YES 1  
 b) #define NO 0  
 c) #include "common.h"  
 d) #line 3000  
 e) #if defined(TRUE)  
     #undef TRUE  
     #define TRUE 1  
   #endif  
 f) #ifdef TRUE

```

        #undef TRUE
        #define TRUE 1
    #endif
g) #if ACTIVE
        #define INACTIVE 0
    #else
        #define INACTIVE 1
    #endif
h) #define CUBE_VOLUME( x ) ( ( x ) * ( x ) * ( x ) )

```

## 练习

- 17.4 编写一个程序，定义一个计算球体积的带参数的宏。在程序中计算半径从1到10的球的体积，并以表格形式打印出结果。球体积计算公式是：

$$(4/3) * \pi * r^3$$

表达式中， $\pi$ 取3.14159。

- 17.5 编写一个程序，输出如下结果：

```
The sum of x and y is 13
```

在程序中定义带有两个参数x和y的宏SUM，用SUM来产生输出结果。

- 17.6 编写一个程序，用宏MINIMUM2计算两个数值中的最小值（数据从键盘输入）。
- 17.7 编写一个程序，用宏MINIMUM3计算三个数值中的最小值，宏MINIMUM3要用到在练习17.6中定义的宏MINIMUM2（数据从键盘输入）。
- 17.8 编写一个程序，用宏PRINT打印出字符串的值。
- 17.9 编写一个程序，用宏PRINTARRAY打印出一个整数数组，数组以及数组元素个数作为宏的参数。
- 17.10 编写一个程序，用宏SUMARRAY计算数值型数组中的各元素值之和，数组以及数组元素个数作为宏的参数。
- 17.11 将17.4到17.10的答案改写成内联函数。
- 17.12 对下列宏，指定预处理器展开宏时可能遇到的问题（如果有）：
- #define SQR( x ) x \* x
  - #define SQR( x ) ( x \* x )
  - #define SQR( x ) ( x ) \* ( x )
  - #define SQR( x ) ( ( x ) \* ( x ) )

## 第 18 章 C 语言遗留代码问题

### 教学目标

- 把键盘输入重定向为来自文件的输入和把屏幕输出重定向到文件中
- 编写变长参数表的函数
- 处理命令行参数
- 处理程序中的意外事件
- 用 C 语言式动态内存分配为数组动态分配内存
- 用 C 语言式动态内存分配改变动态分配的内存大小

### 18.1 简介

本章介绍几个高级话题,入门教程通常不会涉及这些内容。本章介绍的许多内容与特定的操作系统(尤其是 UNIX 和 DOS 系统)有关。C++ 程序员了解这些材料有助于处理 C 语言遗留代码。

### 18.2 UNIX 和 DOS 系统中的输入/输出重定向

通常情况下,程序的输入来自于键盘(标准输入),输出显示在屏幕(标准输出)上。在大多数计算机系统上(特别是 UNIX 和 DOS 系统),重定向来自文件(而不是键盘)的输入、以及把输出重定向到文件中(而不是屏幕上)都是可能的,实现这两种重定向并不需要使用标准库中的文件处理函数。

重定向来自 UNIX 命令行的输入和输出有多种方法。假定有一个可执行文件 `sum`,它需要输入一些整数,每次输入一个,然后不断地计算所输入值的和,在输入文件结束符后结束输入并打印出计算结果。这些整数以及表示输入结束的文件结束符组合键通常都从键盘输入的。利用输入重定向,输入可以存储在一个文件中。例如,假如数据存储存储在文件 `input` 中,那么如下的命令行可以使程序 `sum` 得以执行:

```
$ sum < input
```

输入重定向符(`<`)表示把文件 `input`(而不是键盘)中的数据用作程序的输入。在 DOS 系统中执行输入重定向的方式也一样。

注意,`$`是 UNIX 命令行提示符(某些 UNIX 系统使用提示符`%`)。学生们经常感到这很难理解,为什么重定向是操作系统而不是 C++ 语言的一项功能。

使输入重定向的第二种方法是建立管道。管道(`|`)可以使一个程序的输出重定向为另一个程序的输入。假如程序 `random` 的输出是一系列化随机整数,那么这些输出可以通过管道重定向到程序 `sum` 中。UNIX 系统所用的命令行是:

```
$ random | sum
```

这条命令可以计算出 `random` 所产生的整数值之和。UNIX 和 DOS 系统中都可以建立管道。

可以用输出重定向符 (`>`) 把程序的输出重定向到文件中 (UNIX 和 DOS 中都用这个符号)。例如, 如下的命令行可把程序 `random` 的输出重定向到文件 `out` 中:

```
$ random > out
```

此外, 使用输出添加符 (`>>`) 可以把程序的输出添加到某个现有文件的尾部 (UNIX 和 DOS 中都用这个符号)。例如, 如下所示的命令行可把程序 `random` 的输出添加到用上述命令行建立的文件 `out` 中:

```
$ random >> out
```

### 18.3 变长参数表

(注意: C++ 程序员了解这些材料有助于处理 C 语言遗留代码。C++ 中, 可以用函数重载完成大多数变长参数表的大多数功能)。建立参数个数不定的函数是可能的。该函数原型中的省略号 (...) 表示这个函数接收个数不定的任何类型的参数。注意, 省略号必须放在参数表的最后。变长参数头文件 `stdarg.h` 中的宏和定义 (见图 18.1) 可用来建立变长参数表。

标识符	解释
<code>va_list</code>	用来保存宏 <code>va_start</code> 、 <code>va_arg</code> 和 <code>va_end</code> 所需信息的一种类型。为了访问变长参数表中的参数, 必须声明 <code>va_list</code> 类型的一个对象
<code>va_start</code>	访问变长参数表中的参数之前调用的宏, 该宏初始化用 <code>va_list</code> 声明的对象, 初始化结果供 <code>va_arg</code> 和 <code>va_end</code> 使用
<code>va_arg</code>	展开成一个表达式的宏, 该表达式具有变长参数表中下一个参数的值和类型。每次调用 <code>va_arg</code> 都会修改用 <code>va_list</code> 声明的对象, 从而使该对象指向参数表中的下一个参数
<code>va_end</code>	该宏使程序能够从用宏 <code>va_start</code> 引用变长参数表的函数中正常返回

图 18.1 在头文件 `stdarg.h` 中定义的类型和宏

图 18.2 中的程序演示了接收可变参数个数的函数 `average`。函数 `average` 的第一个参数是要求平均值的数据个数。

```
1 // Fig. 18.2: fig18_02.cpp
2 // Using variable-length argument lists
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdarg.h>
6
7 double average( int, ... );
8
9 int main()
10 {
11     double w = 37.5, x = 22.5, y = 1.7, z = 10.2;
12
13     cout << setiosflags( ios::fixed | ios::showpoint )
14          << setprecision( 1 ) << "w = " << w << "\nx = " << x
15          << "\ny = " << y << "\nz = " << z << endl;
16     cout << setprecision( 3 ) << "\nThe average of w and x is "
```

```
17         << average( 2, w, x )
18         << "\nThe average of w, x, and y is "
19         << average( 3, w, x, y )
20         << "\nThe average of w, x, y, and z is "
21         << average( 4, w, x, y, z ) << endl;
22     return 0;
23 }
24
25 double average( int i, ... )
26 {
27     double total = 0;
28     va_list ap;
29
30     va_start( ap, i );
31
32     for ( int j = 1; j <= i; j++ )
33         total += va_arg( ap, double );
34
35     va_end( ap );
36
37     return total / i;
38 }
```

**输出结果：**

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2
```

```
The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

图 18.2 变长参数表的用法

函数 `average` 使用了头文件 `stdarg.h` 中所有的定义和宏。宏 `va_start`、`va_arg` 和 `va_end` 用 `va_list` 类型的对象 `ap` 处理函数 `average` 的变长参数表。函数先调用宏 `va_start` 初始化供 `va_arg` 和 `va_end` 使用的对象 `ap`。宏 `va_start` 有两个参数，一个是对象 `ap`，另一个是变长参数表中省略号前的最右边参数的标识符（本例中为 `i`，`va_start` 用 `i` 确定变长参数表的起始位置）。然后，函数 `average` 反复把变长参数表中的参数与变量 `total` 相加，加到 `total` 中的值是调用宏 `va_arg` 从变长参数表中检索到的。宏 `va_arg` 有两个参数，一个是对象 `ap`，另一个是要从参数表中接收的值的类型（本例为 `double`），该宏返回这个参数的值。宏 `va_end` 只有对象 `ap` 这一个参数。函数 `average` 调用宏 `va_end` 使得程序从 `average` 正常返回到函数 `main`。最终，函数计算了平均值并返回到了函数 `main`。注意我们只对参数表的变长部分使用 `double` 类型的参数。实际上只要每次使用 `va_arg` 时都使用正确的类型，就可以使用任何数据类型或任何数据类型的组合。

**常见编程错误 18.1**

把省略号放在函数参数表中间。省略号只能放在参数表的最后。

## 18.4 使用命令行参数

许多操作系统（特别是 DOS 和 UNIX）能够把命令行参数传递给参数表中包含参数 `int argc` 和 `char * argv[]` 的 `main` 函数。参数 `argc` 是命令行参数的个数，`argv` 是存储实际命令行参数的字符串数组。命令行参数常用来打印参数、给程序传递选项和文件名。

图 18.3 中的程序把一个文件复制到另一个文件中（一次复制一个字符）。程序的可执行文件称为 `copy`。在 UNIX 系统中，运行程序 `copy` 的典型的命令行为：

```
$ copy input output
```

该命令行表示把文件 `input` 复制到文件 `output` 中。当执行该程序时，如果 `argc` 不等于 3（`copy` 也算一个参数），程序就打印出错误消息并终止执行，否则数组 `argv` 中将包含字符串 `"copy"`、`"input"` 和 `"output"`。程序将该命令行的第二个和第三个参数用作文件名。这两个文件通过创建 `ifstream` 的对象 `inFile` 和 `ofstream` 的对象 `outFile` 打开。如果成功地打开了这两个文件，那么就把文件 `input` 中的字符逐个复制到文件 `output` 中，直到遇到为文件 `input` 设置的文件结束符为止。然后程序终止执行，其结果是获得与文件 `input` 完全一样的副本。注意，并非所有的计算机系统都像 UNIX 和 DOS 那样能够容易地支持命令行参数。例如，Macintosh 和 VMS 系统要求用专门的设置来处理命令行参数。有关命令行参数的更详细的情况，参见系统手册。

```
1 // Fig. 18.3: fig18_03.cpp
2 // Using command-line arguments
3 #include <iostream.h>
4 #include <fstream.h>
5
6 int main( int argc, char *argv[] )
7 {
8     if ( argc != 3 )
9         cout << "Usage: copy infile outfile" << endl;
10    else {
11        ifstream inFile( argv[ 1 ], ios::in );
12        if ( !inFile )
13            cout << argv[ 1 ] << " could not be opened" << endl;
14
15        ofstream outFile( argv[ 2 ], ios::out );
16        if ( !outFile )
17            cout << argv[ 2 ] << " could not be opened" << endl;
18
19        while ( !inFile.eof() )
20            outFile.put( static_cast< char >( inFile.get() ) );
21    }
22
23    return 0;
24 }
```

图 18.3 命令行参数的用法

## 18.5 对编译多个源文件程序的说明

前面已经讲过,建立包含多个源文件的程序是可能的(见第6章“类和数据抽象”)。在建立包含多个文件的程序时,有几点是要考虑的。例如,函数的定义必须完整地存在于同一个文件中,而不能把它分散在两个或多个文件中。

第3章介绍了存储类别和范围的概念。我们知道,在所有函数定义之外声明的变量在默认情况下具有静态存储类别,这种变量称为“全局变量”。全局变量能够被同一个文件中该变量声明之后的所有函数访问。其他文件中的函数也能够访问全局变量,但是必须在使用该全局变量的每一个文件中予以声明。例如,如果在一个文件中定义了全局变量 `flag`,并在第二个文件中引用了该变量,那么第二个文件在使用该变量之前必须做如下声明:

```
extern int flag;
```

在这条声明语句中,存储类别说明符 `extern` 告诉编译器:变量 `flag` 或者稍后定义在同一个文件中,或者在另一个文件中定义。编译器就会通知连接程序:该程序没能解决在该文件中出现的对变量 `flag` 的引用(编译器并不知道变量 `flag` 定义在何处,因此让连接程序查找 `flag`)。如果连接程序没有找到对 `flag` 的定义,那么它就会发出错误消息并且不生成可执行文件。如果连接程序找到了一个全局变量 `flag` 的正确定义,它就会指明其位置,从而解决对该变量的引用。

### 性能提示 18.1

因为全局变量能够被所有函数访问,没有给函数传递数据的开销,所以使用全局变量提高了程序的性能。

### 软件工程视点 18.1

全局变量违背了最低访问权限原则,除非应用程序的性能至关重要,否则不要使用全局变量。

就像用 `extern` 声明的全局变量能够被其他程序文件使用一样,函数原型也能把函数的范围扩展到定义该函数的文件之外(函数原型中不必使用说明符 `extern`),只要在使用该函数的每一个文件中包含该函数的函数原型,然后再一起编译这些文件(见17.2节)。函数原型告诉编译器:指定的函数或者稍后定义在同一个文件中,或者定义在另一个文件中。同样,编译器并不试图解决对这种函数的引用,而是把该任务留给连接程序。如果连接程序不能定位正确的函数定义,它就会发出错误消息。

举一个用函数原型扩展函数范围的例子。对于包含预处理指令 `#include < string.h >` 的程序来说,这条预处理指令把 `strcmp` 和 `strcat` 等函数的函数原型包含到该文件中,该文件中的其他函数可以用函数 `strcmp` 和 `strcat` 来完成其任务。函数 `strcmp` 和 `strcat` 是单独定义的,我们不需要知道它们定义在何处,只要在程序中使用其代码就行了。连接程序会自动解决对这些函数的引用。这种处理方式可以让我们使用标准库中的函数。

### 软件工程视点 18.2

建立包含多个源文件的程序能够提高软件的复用性和良好的软件工程。函数可以被多个应用程序共用。在这种情况下,这些函数应该保存在它们自己的源文件中,并且每个源文件都应该包括含有函数原型的相应的头文件。这样,只要包含了合适的头文件,并把应用程序和相应的源文件一起编译,各个应用程序的程序员就都可以使用同一段代码。



### 可移植性提示 18.1

某些系统不支持超过 6 个字符的全局变量名或函数名，在编写要在多个平台上运行的程序时要注意这一点。

可以把全局变量或函数的范围限制在定义它的文件中。对全局变量或函数使用存储类别说明符 `static` 就可以防止定义在其他几个文件中的函数（没有在同一个文件中定义）使用这些全局变量或函数。这种连接方式称为“内部连接”（`internal linkage`）。在定义全局变量和函数时，如果没有在全局变量和函数名的前面使用 `static`，那么这些全局变量和函数具有“外部连接”（`extern linkage`）属性。如果在其他文件中包含了正确地声明和（或）函数原型，那么能够在这些文件中访问到这些全局变量和函数。

如下的全局变量声明建立了 `float` 类型的变量 `pi`，该变量初始化为 3.14159，并且只能被同一个文件中的函数访问到：

```
static float pi = 3.14159;
```

说明符 `static` 通常和工具函数一起使用，这些函数只被某个特定文件中的函数访问。如果某个函数不需要在特定文件之外使用，那么应该用 `static` 关键字强制实现最低访问权限原则。如果一个函数在使用前定义，那么应该用 `static` 定义该函数，否则应该把 `static` 用于函数原型。

在建立包含多个源文件的大型程序时，因为一个小小的改动而要编译整个程序是一件令人头疼的事。为了只编译改动过的源文件，许多系统都提供了专用的实用程序。在 UNIX 系统中，这种实用程序称为 `make`。`make` 通过读取称为 `makefile` 的文件来编译和连接改动过的程序（`makefile` 中包含了编译和连接该程序的指令）。在 PC 机上使用的系统如 Borland C++ 和 Microsoft Visual/C++ 也都提供了实用程序 `make`。有关实用程序 `make` 的更详细的情况，参见特定系统的手册。

## 18.6 用 `exit` 和 `atexit` 终止程序的执行

通用实用程序库（`stdlib.h`）提供了终止程序执行（并非从 `main` 函数返回）的方法。函数 `exit` 强制终止程序的执行，通常在程序检测到输入错误或不能打开某个要处理的文件时使用该函数。函数 `atexit` 用来注册在程序正常终止时要被调用的函数，也就是运行完函数 `main` 或调用 `exit` 时都会调用该函数。

函数 `atexit` 的参数是一个指向函数的指针（即函数名）。在程序终止时所要调用的函数不能有参数和返回值，并且在程序终止时最多只能注册 32 个这样的函数。

函数 `exit` 只有一个参数，通常是符号常量 `EXIT_SUCCESS` 或 `EXIT_FAILURE`。如果用 `EXIT_SUCCESS` 调用函数 `exit`，那么就把表示成功执行完毕的实现时定义值返回给调用环境。如果用 `EXIT_FAILURE` 调用函数 `exit`，那么就把表示执行不成功的实现时定义值返回给调用环境。在调用函数 `exit` 时，在此之前用 `atexit` 注册的函数以注册相反的次序被调用，所有与该程序关联的流都被刷新和关闭，然后把控制交给运行环境。图 18.4 中的程序测试了函数 `exit` 和 `atexit`。程序提示用户是用 `exit` 终止程序的执行还是通过运行完函数 `main` 来终止程序的执行。注意，在这两种情况下，函数 `print` 都会在程序终止时得到执行。

```
1 // Fig. 18.4: fig18_04.cpp
2 // Using the exit and atexit functions
```

```
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 void print( void );
7
8 int main()
9 {
10     atexit( print );          // register function print
11     cout << "Enter 1 to terminate program with function exit"
12           << "\nEnter 2 to terminate program normally\n";
13
14     int answer;
15     cin >> answer;
16
17     if ( answer == 1 ) {
18         cout << "\nTerminating program with function exit\n";
19         exit( EXIT_SUCCESS );
20     }
21
22     cout << "\nTerminating program by reaching the of main"
23           << endl;
24
25     return 0;
26 }
27
28 void print( void )
29 {
30     cout << "Executing function print at program termination\n"
31           << "Program terminated" << endl;
32 }
```

**输出结果:**

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
:1
```

```
Terminating program with function exit
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
:2
```

```
Terminating program by reaching end of main
Executing function print at program termination
Program terminated
```

图 18.4 函数 exit 和 atexit 的用法

## 18.7 volatile 类型限定符

volatile 类型限定符用于定义可以从程序之外改变的变量(即不完全由程序控制的变量)。这样,

编译器不能根据执行速度、内存使用等指标进行优化,因为优化要求变量行为只受编译器所知程序活动的影响。

## 18.8 整数和浮点数常量的后缀

为了指定某个常量是整数类型还是浮点数类型, C 语言提供了整数和浮点数常量的后缀。整数后缀包括代表无符号整数类型 (unsigned) 的 u 或 U、代表长整数类型 (long) 的 l 或 L 以及代表无符号长整数类型 (unsigned long) 的 ul 或 UL。如下的常量分别代表 unsigned、long 和 unsigned long 类型的常量:

```
174u
8358L
28373ul
```

如果某个整数常数不带后缀,那么其类型是能够存储该值的第一个类型(类型顺序: int、long int 和 unsigned long int)。

浮点数常量的后缀包括代表 float 类型的 f 或 F 和代表 long double 类型的 l 或 L。如下的常量分别代表 long double 和 float 类型的常量:

```
3.14159L
1.28f
```

不带后缀的浮点数常量自动是 double 类型。

## 18.9 信号处理

一个没有预料到的事件(即信号)会使程序过早地终止。这些事件包括中断(在 UNIX 或 DOS 系统中键下<ctrl>c)、非法的指令、段访问非法、来自操作系统的终止命令以及浮点数异常(除以 0 或乘以大型浮点数)等等。信号处理库中的函数 signal 能够捕捉到没有预料到的事件。该函数有两个参数,一个是整数信号值,另一个是指向信号处理函数的指针。信号也可以用函数 raise 产生,该函数的参数是一个整数信号值。图 18.5 列出了在头文件 signal.h 中定义的标准信号。图 18.6 中的程序演示了函数 signal 和 raise 的用法。

信号	解释
SIGABRT	程序异常终止(如调用函数 abort)
SIGFPE	错误算术运算,如除以 0 或运算结果溢出
SIGILL	检测非法指令
SIGINT	接受交互式信号
SIGSEGV	非法访问内存
SIGTERM	发送给程序的终止请求

图 18.5 在头文件 signal.h 中定义的标准信号

图 18.6 中的程序用函数 signal 捕捉交互式信号(SIGINT)。程序先用 SIGINT 和指向函数 signal\_handler 的指针调用函数 signal(函数名是指向该函数在内存中起始位置的指针)。当产生

SIGINT 类型的信号时, 调用函数 `signal_handler`, 该函数打印出一条消息并给出能够使程序继续正常执行的选项。如果用户想继续执行程序, 那么就通过调用函数 `signal` 重新初始化信号处理函数(某些系统要求重新初始化信号处理函数), 然后把控制返回到程序中检测到该信号的位置。在示例的程序中, 函数 `raise` 用来模拟交互式信号。程序中产生一个在 1 到 50 之间的随机整数, 如果这个数等于 25, 就调用函数 `raise` 产生该信号。交互式信号通常来自程序之外, 例如在 UNIX 或 DOS 系统中, 用户在程序执行期间键入 `<ctrl>c` 就会产生一个终止程序执行的交互式信号。信号处理能够用来捕捉这种交互式信号并防止程序终止执行。

```
1 // Fig. 18.6: fig18_06.cpp
2 // Using signal handling
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <signal.h>
6 #include <stdlib.h>
7 #include <time.h>
8
9 void signal_handler( int );
10
11 int main()
12 {
13     signal( SIGINT, signal_handler );
14     srand( time( 0 ) );
15
16     for ( int i = 1; i < 101; i++ ) {
17         int x = 1 + rand() % 50;
18
19         if ( x == 25 )
20             raise( SIGINT );
21
22         cout << setw( 4 ) << i;
23
24         if ( i % 10 == 0 )
25             cout << endl;
26     }
27
28     return 0;
29 }
30
31 void signal_handler( int signalValue )
32 {
33     cout << "\nInterrupt signal (" << signalValue
34         << ") received.\n"
35         << "Do you wish to continue (1 = yes or 2 = no)? ";
36
37     int response;
38     cin >> response;
39
40     while ( response != 1 && response != 2 ) {
41         cout << "(1 = yes or 2 = no)? ";
42         cin >> response;
43     }
44 }
```

```

45     if ( response == 1 )
46         signal( SIGINT, signal_handler );
47     else
48         exit( EXIT_SUCCESS );
49 )

```

输出结果:

```

 1      2      3      4      5      6      7      8      9     10
11     12     13     14     15     16     17     18     19     20
21     22     23     24     25     26     27     28     29     30
31     32     33     34     35     36     37     38     39     40
41     42     43     44     45     46     47     48     49     50
51     52     53     54     55     56     57     58     59     60
61     62     63     64     65     66     67     68     69     70
71     72     73     74     75     76     77     78     79     80
81     82     83     84     85     86     87     88
Interrupt signal (4) received.
Do you wish to continue ( 1 = yes or 2 = no )? 1
89     90
91     92     93     94     95     96     97     98     99    100

```

图 18.6 信号处理的使用

## 18.10 动态内存分配：函数 calloc 和 realloc

第 7 章介绍了 C++ 动态内存分配使用 new 和 delete，并比较了 new 和 delete 与 C 语言函数 malloc 和 free。C++ 程序员应使用 new 和 delete 而不用 malloc 和 free。但大多数 C++ 程序员都要面对 C 语言遗留代码，因此下面要介绍 C 语言遗留代码中的动态内存分配。

通用实用程序库 (stdlib.h) 提供了另外两个动态分配内存的函数 calloc 和 realloc。这两个函数可用来建立和修改动态数组。正如第 5 章“指针与字符串”所介绍的，指向数组的指针可以像数组那样建立下标。因此，对于指向用函数 calloc 建立的连续存储区的指针，我们可以像操作数组那样操作这个指针。函数 calloc 的函数原型为：

```
void *calloc (size_t nmemb, size_t size);
```

该函数接收两个参数（一个是元素个数 nmemb，另一个是每一个元素的大小 size），并把数组元素初始化为 0。函数返回指向所分配内存的指针（如果分配内存不成功则返回空（0）指针）。

函数 realloc 修改由调用 malloc、calloc 或 realloc 函数所分配的对象的大小。如果所分配的内存数量大于以前分配的内存数量，原始对象的内容不再修改。否则，就不改变原始对象中新对象大小范围内的内容。函数 realloc 的函数原型为：

```
void *realloc(void * ptr, size_t size);
```

该函数接收两个参数，一个是指向原始对象的指针（ptr），另一个是该对象的新的 size。如果 ptr 为 0，那么 realloc 等价于 malloc。如果 size 为 0 并且 ptr 不等于 0，那么就释放对象所占用的内存。否则，如果 ptr 不等于 0 并且 size 大于 0，那么 realloc 就试图为该对象分配新的内存块。如果分配不成功，ptr 所指向的对象不再修改。函数 realloc 或者返回指向重新分配的内存的指针，或者返回一个空指针。

## 18.11 无条件转移：goto 语句

本书自始至终都强调用结构化程序设计技术建立易于调试、维护和修改的可靠的软件。在某些情况下，性能比严格的结构化程序设计方法更重要。这时可以使用某些非结构化的程序设计方法。例如，可以在继续循环的条件为假之前用break语句终止循环结构的执行。如果在循环终止之前已经完成了任务，这样做可省去不必要的循环过程。

另一个非结构化程序设计的实例是使用无条件转移的goto语句。goto语句把程序的控制流转移到在goto语句中指定的标号之后的第一条语句。标号是跟有冒号的标识符，它必须与引用它的goto语句在同一个函数中。图18.7中的程序用goto语句执行了10次循环并在每次循环中打印出了计数器的值。程序先把count初始化为1，然后测试count是否大于10（因为标号不执行任何动作，所以程序跳过标号start）。如果count大于10，控制就从goto语句转移到标号end后的第一条语句，否则就打印出count的值，并把count加1，然后再把控制转移到标号start之后的第一条语句。

```
1 // Fig. 18.7: fig18_07.cpp
2 // Using goto
3 #include <iostream.h>
4
5 int main()
6 {
7     int count = 1;
8
9     start:                // label
10        if ( count > 10 )
11            goto end;
12
13        cout << count << " ";
14        ++count;
15        goto start
16
17     end:                  // label
18        cout << endl;
19
20    return 0;
21 }
```

**输出结果：**

1 2 3 4 5 6 7 8 9 10

图 18.7 goto 语句的使用

第2章曾说过，只要有三种控制结构（顺序结构、选择结构和循环结构）就可以编写任何程序。遵循结构化程序设计规则可能会建立一个深层的难以从中退出的嵌套式控制结构，某些程序员在这种情况下用goto语句来快速地退出深层嵌套式结构，这样就不用再通过测试许多条件来退出控制结构。

### 性能提示 18.2

goto语句可有效地退出深层嵌套式控制结构。

### 软件工程视点 18.3

goto语句应该只用于面向性能的应用程序中。goto语句是非结构化语句，使得程序难以调试、维护和修改。

## 18.12 联合体

联合体(用关键字union定义)也是一块内存区域,多数情况下,联合体可以包含多种数据类型。同一个时候只能引用一个成员(其成员共享了同一个存储空间)。保证用正确数据类型的成员名引用联合体中的数据是程序员的责任。

### 常见编程错误 18.2

引用联合体的成员而不是最后一个存储的成员的結果是不确定的,会将这个存储数据作为另一种类型。

### 可移植性提示 18.2

如果数据在联合体中存储时作为一种类型而在引用时作为另一种类型,那么结果与实现过程有关。

程序执行中的变量无非是两种情况:某些对象是不相关的,而其他一些对象是相关的。联合体用来使相关对象共享存储空间而不是在不使用的对象上浪费空间。联合体的成员可以是任何数据类型。用来存储联合体的字节数至少要能够足以存储最大的成员。

### 性能提示 18.3

用联合体节省内存。

### 可移植性提示 18.3

存储联合体所需的存储空间是与实现相关的。

### 可移植性提示 18.4

有些联合体可能不容易移植到其他计算机系统。联合体是否可移植取决于该系统中联合体成员数据的存储定位要求。

联合体用关键字union声明,其格式与结构和类的声明是一样的。声明语句:

```
union Number {  
    int x;  
    float;  
};
```

表示Number是具有成员int x和float y的联合体类型。联合体通常定义在程序的main函数之前,因而能够被程序中所有的函数用来声明变量。

### 软件工程视点 18.4

和结构的声明一样,联合体的声明仅仅是建立一种新的数据类型。把联合体和结构的声明放在所有函数之外不能建立全局变量。

可以对联合体执行的操作有:把该联合体赋给另一个具有相同类型的联合体。取地址(&)、以及用结构成员运算符(.)和结构指针运算符(->)访问联合体的成员。正像结构不能比较大小一样,联合体由于同样的原因也不能比较大小。

### 常见编程错误 18.3

因为不同的系统对联合体的存储方式不同,所以比较联合体是一种语法错误。

联合体与类相似,可以用构造函数初始化任何成员。联合体不带构造函数时,可以用相同类型的另一个联合体初始化,用联合体第一个成员类型的表达式初始化,或用花括号中联合体第一个成

员类型的初始化值初始化。联合体可以有其他成员函数，如析构函数，但联合体的成员函数不能声明为虚函数。联合体的成员默认为 `public`。

#### 常见编程错误 18.4

在声明中初始化联合体的值的类型与联合体中第一个成员的类型不同。

联合体不能用作继承中的基类，即不能从联合体派生类。联合体可以用对象成员，但这些对象不能有构造函数、析构函数或重载的赋值运算符。联合体的数据成员不能声明为 `static`。

图 18.8 中的程序用类型为 `union number` 的变量 `value` 显示存储在联合体中的 `int` 类型和 `float` 类型的值。程序的输出结果与实现过程有关。该程序的输出结果表明：`float` 类型的值的内部表示与 `int` 类型的值的内部表示大不相同。

```
1 // Fig. 18.8: fig18_08.cpp
2 // An example of a union
3 #include <iostream.h>
4
5 union Number {
6     int x;
7     float y;
8 };
9
10 int main()
11 {
12     Number value;
13
14     value.x = 100;
15     cout << "Put a value in the integer member\n"
16           << "and print both members.\nint:  "
17           << value.x << "\nfloat: " << value.y << "\n\n";
18
19     value.y = 100.0;
20     cout << "Put a value in the floating member\n"
21           << "and print both members.\nint:  "
22           << value.x << "\nfloat: " << value.y << endl;
23     return 0;
24 }
```

#### 输出结果：

```
Put a value in the integer member
and print both members
int: 100
float: 3.504168e-16
```

```
Put a value in the integer member
and print both members
int: 0
float: 100
```

图 18.8 以两种不同的成员数据类型打印联合体的值

匿名联合体是没有类型名的联合体，在终止分号之前不定义对象和指针。这种联合体不生成类型，但生成无名对象。匿名联合体的成员可以在声明该匿名联合体的范围中直接访问，和任何其他局部变量一样，不需用圆点 (.) 或箭头 (->) 运算符。



匿名联合体有一些限制。匿名联合体只能包含数据成员。匿名联合体的所有成员应为 public。匿名联合体在全局范围内声明时（即在文件范围）应显式声明为 static。图 18.9 演示匿名联合体的用法。

```
1 // Fig. 18.9: fig18_09.cpp
2 // Using an anonymous union
3 #include <iostream.h>
4
5 int main()
6 {
7     // Declare an anonymous union.
8     // Note that members b, d, and f share the same space.
9     union {
10         int b;
11         double d;
12         char *f;
13     }
14
15     // Declare conventional local variables
16     int a = 1;
17     double c = 3.3;
18     char *e = "Anonymous";
19
20     // Assign a value to each union member
21     // successively and print each.
22     cout << a << ' ';
23     b = 2;
24     cout << b << endl;
25
26     cout << c << ' ';
27     d = 4.4;
28     cout << d << endl;
29
30     cout << e << ' ';
31     f = "union";
32     cout << f << endl;
33
34     return 0;
35 }
```

**输出结果：**

```
1 2
3.3 4.4
Anonymous union
```

图 18.9 匿名联合体的用法

## 18.13 连接指定

从 C++ 程序调用 C 编译器编写和编译的函数是可能的。正如 3.20 节所讲到的，为了进行类型安全的连接，C++ 对函数名作了专门的编码。然而，C 没有对函数名编码。因为 C++ 代码要求专门编码的函数名，所以在试图把 C 代码与 C++ 代码连接时，在 C 中编译的函数不能被识别。为了告诉 C++ 编译器某个函数是在 C 编译器上编译的以及防止函数名被 C++ 编译器编码，C++ 允许程序员提

供连接指定。当开发出大型专用的函数库后,如果用户无法获得要在C++上重新编译的源代码,或者没有时间把函数库从C转为C++,这时连接指定是非常有用的。

要通知编译器一个或多个函数已经在C中编译完,写出其函数原型:

```
extern "C" 函数原型 // 单个函数
extern "C" // 多个函数
{
    函数原型
}
```

这些声明通知编译器指定的函数不在C++中编译,因此不对连接说明中列出的函数名编码,这些函数能够得到正确的连接。C++环境通常包含了C标准库,并且不要求程序员对这些函数使用连接指定。

## 小结

- 在许多计算机系统上(特别是UNIX和DOS系统),把输入重定向到某个程序以及重定向来自程序的输出都是可能的。在UNIX和DOS命令行上用输入重定向符(<)或管道(|)可以把输入重定向。在UNIX和DOS命令行上用输出重定向符(>)或输出添加符(>>)可把输出重定向。输出重定向符把程序的输出存储在一个文件中,输出添加符把输出添加到文件末尾。
- 变长参数头文件stdarg.h中的宏和定义能用来建立具有变长参数表的函数。
- 函数原型中的省略号(...)表示参数的个数是不定的。
- 类型va\_list用来保存宏va\_start、va\_arg和va\_end所需信息的一种类型。为了访问变长参数表中的参数,必须声明va\_list类型的一个对象。
- va\_start是在访问变长参数表中的参数之前使用的宏,该宏初始化用va\_list声明的对象,初始化结果供宏va\_arg和va\_end使用。
- 宏va\_arg展开成一个表达式,该表达式具有变长参数表中下一个参数的值和类型。每次调用va\_arg都会修改用va\_list声明的对象,从而使该对象指向参数表中的下一个参数。
- va\_end使得变长参数表用宏va\_list引用的函数能够正常返回。
- 许多操作系统(特别是DOS和UNIX)能够把命令行参数传递给参数表中包含参数int argc和char\* argv[]的main函数。参数argc是命令行参数的个数,argv是存储实际命令行参数的字符串数组。
- 函数的定义必须完整地存在于同一个文件中,而不能把它分散在两个或多个文件中。
- 全局变量必须在用到它的每一个文件中声明。
- 函数原型能把函数的范围扩展到定义该函数的文件之外(函数原型中不必使用说明符extern),只要在使用该函数的每一个文件中包含该函数的函数原型,然后再一起编译这些文件就可以到达目的。
- 对全局变量或函数使用存储类别说明符static就可以防止定义在其他文件中的函数使用这些全局变量或函数。这种连接方式称为“内部连接”。在定义全局变量和函数时,如果没有在全局变量和函数名的前面使用static,那么这些全局变量和函数具有“外部连接”属性。如果在其他文件中包含了正确的声明和(或)函数原型,那么能够在这些文件中访问到这些全局变量和函数。

- 说明符 `static` 通常和工具函数一起使用, 这些函数只被某个特定文件中的函数访问。如果某个函数不需要在特定文件之外使用, 那么应该用 `static` 关键字强制实现最低访问权限原则。
- 在建立包含多个源文件的大型程序时, 因为一个小小的改动而要编译整个程序是一件令人头疼的事。为了只编译改动过的源文件, 许多系统都提供了专门的实用程序。在 UNIX 系统中, 这种实用程序称为 `make`。`make` 通过读取称为 `makefile` 的文件来编译和连接改动过的程序 (`makefile` 中包含了编译和连接该程序的指令)。
- 函数 `exit` 强迫终止程序的执行。
- 函数 `atexit` 用来注册在程序正常终止时要被调用的函数, 也就是运行完函数 `main` 或调用 `exit` 时都会调用该函数。
- 函数 `atexit` 的参数是一个指向函数的指针 (即函数名)。在程序终止时所要调用的函数不能有参数和返回值, 并且最多只能注册 32 个这样的函数。
- 函数 `exit` 只有一个参数, 通常是符号常量 `EXIT_SUCCESS` 或 `EXIT_FAILURE`。如果用 `EXIT_SUCCESS` 调用函数 `exit`, 那么就把表示成功执行完毕的实现时定义值返回给调用环境。如果用 `EXIT_FAILURE` 调用函数 `exit`, 那么就把表示执行不成功的实现时定义值返回给调用环境。
- 在调用函数 `exit` 时, 在此之前用 `atexit` 注册的函数以注册相反的次序被调用, 所有与该程序关联的流都被刷新和关闭, 然后把控制交给运行环境。
- 限定符 `volatile` 用来禁止变量的优化, 因为该变量可以在程序范围之外修改。
- 为了指定某个常量是整数类型还是浮点数类型, C++ 语言提供整数和浮点数常量的后缀。整数后缀包括代表无符号整数类型的 `u` 或 `U`、代表长整数类型的 `l` 或 `L`、以及代表无符号长整数类型的 `ul` 或 `UL`。如果某个整数常量不带后缀, 那么其类型是能够存储该值的第一个类型 (类型顺序: `int`、`long int` 和 `unsigned long int`)。无后缀的浮点数常量自动是 `double` 类型。
- 信号处理库中的函数 `signal` 能够捕捉到没有预料到的事件。该函数有两个参数, 一个是整数信号值, 另一个是指向信号处理函数的指针。
- 信号也可以用函数 `raise` 产生, 该函数的参数是一个整数信号值。
- 通用实用程序库 (`stdlib.h`) 提供了另外两个动态分配内存的函数 `calloc` 和 `realloc`。这两个函数可用来建立和修改动态数组。
- 函数 `calloc` 接收两个参数 (一个是元素个数 `nmemb`, 另一个是每一个元素的大小 `size`), 并把数组元素初始化为 0。函数返回指向所分配内存的指针 (如果分配内存不成功则返回 `NULL` 指针)。
- 函数 `realloc` 修改用函数调用 `malloc`、`calloc` 和 `realloc` 所分配的对象的大小 (`size`)。如果所分配的内存数量大于以前分配的内存数量, 就不修改原始对象的内容。
- 函数 `realloc` 接收两个参数, 一个是指向原始对象的指针 (`ptr`), 另一个是该对象的新的尺寸 (`size`)。如果 `ptr` 为 `NULL`, 那么 `realloc` 等价于 `malloc`。如果 `size` 为 0 并且 `ptr` 不等于 `NULL`, 那么就释放对象所占用的内存。否则, 如果 `ptr` 不等于 `NULL` 并且 `size` 大于 0, 那么 `realloc` 就试图为该对象分配新的内存块。如果分配不成功, 那么就不修改 `ptr` 所指向的对象。函数 `realloc` 或者返回指向重新分配的内存的指针, 或者返回一个 `NULL` 指针。
- `goto` 语句把程序的控制流转移到在 `goto` 语句中指定的标号之后的第一条语句。
- 标号是跟有冒号的标识符, 它必须与引用它的 `goto` 语句在同一个函数中。
- 联合体是一种派生的数据类型, 其成员共享同一个存储空间。联合体的成员可以是任何数据类型。为联合体保留的存储空间必须足以存储最大的成员。多数情况下, 联合体中包含两种或多种数据类型。同一个时候只能引用联合体中的一个成员 (一种数据类型)。

- 联合体是用关键字 `union` 声明的，声明方式与声明结构相同。
- 只能用第一个成员类型的值初始化一个联合体。
- 为了告诉 C++ 编译器某个函数是在 C 编译器上编译以及防止函数名被 C++ 编译器编码，C++ 允许程序员提供连接指定。
- 要通知编译器一个或多个函数已经在 C 中编译完，则写出其函数原型：

```
extern "C" 函数原型 // 单个函数
extern "C" // 多个函数
{
    函数原型
}
```

- 这些声明通知编译器指定的函数不在 C++ 中编译，因此不对连接指定中列出的函数名编码。这些函数能够得到正确的连接。
- C++ 环境通常包含了 C 标准库，并且不要求程序员对这些函数使用连接指定。

## 术语

append output symbol >>	输出添加符	pipe	管道
argv		pipng	建立管道
atexit		raise	
calloc		realloc	
command-line arguments	命令行参数	redirect input symbol <	重定向输入符
const		redirect output symbol >	重定向输出符
dynamic arrays	动态数组	segmentation violation	段违规
event	事件	signal	
exit		signal handling library	信号处理库
external linkage	外部连接	signal.h	
extern storage class specifier	extern 存储类说明符	static storage class specifier	static 存储类说明符
EXIT_FAILURE		stdarg.h	
EXIT_SUCCESS		trap	捕捉
extern "C"		union	
float suffix (f or F)	float 后缀	unsigned integer suffix (u or U)	unsigned 整数后缀
floating-point exception	浮点异常	unsigned long integer suffix (ul or UL)	unsigned long 整数后缀
goto statement	goto 语句	va_arg	
I/O redirection	I/O 重定向	va_end	
illegal instruction	非法指令	va_list	
internal linkage	内部连接	va_start	
interrupt	中断	variable-length argument list	变长参数表
long double suffix (l or L)	long double 后缀	volatile	
long integer suffix (l or L)	long integer 后缀		
make			
makefile			

## 自测练习

### 18.1 填空

- a) \_\_\_\_\_ 符用来把来自键盘的输入重定向成来自文件的输入。
- b) \_\_\_\_\_ 符把屏幕输出重定向到文件中。
- c) \_\_\_\_\_ 符用来把程序的输出添加到文件尾部。
- d) \_\_\_\_\_ 用来把文件的输出重定向成另一个文件的输入。
- e) 函数参数表中的 \_\_\_\_\_ 表示函数接收的参数个数是不定的。
- f) 在访问变长参数表中的参数之前必须先使用宏 \_\_\_\_\_。
- g) 宏 \_\_\_\_\_ 用来访问变长参数表中的参数。
- h) 宏 \_\_\_\_\_ 使得变长参数表用宏 `va_start` 引用的函数能够正常返回。
- i) 函数 `main` 的参数 \_\_\_\_\_ 用来接收命令行参数的个数。
- j) 函数 `main` 的参数 \_\_\_\_\_ 以字符串方式存储命令行参数。
- k) UNIX 实用程序 \_\_\_\_\_ 读取一个称为 \_\_\_\_\_ 的文件, 这个文件中包含了编译和连接由多源文件组成的程序的指令。实用程序只编译自上次编译后做了修改的文件。
- l) 函数 \_\_\_\_\_ 强制程序终止执行。
- m) 函数 \_\_\_\_\_ 注册在程序正常终止时要调用的函数。
- n) 类型限定符 \_\_\_\_\_ 表示对象在初始化后不应该修改。
- o) 为了准确地指明某个常量是整数或浮点数类型, 可把表示该类型的 \_\_\_\_\_ 放在该整数和浮点数常量之后。
- p) 函数 \_\_\_\_\_ 可用来捕捉没有预料到的事件。
- q) 函数 \_\_\_\_\_ 产生一个来自于程序内部的信号。
- r) 函数 \_\_\_\_\_ 动态分配数组的内存, 并把数组的元素初始化为 0。
- s) 函数 \_\_\_\_\_ 改变以前动态分配的内存块的大小。
- t) \_\_\_\_\_ 包含的变量集合在不同时刻占据相同的内存。
- u) \_\_\_\_\_ 关键字用于引用联合体定义。

## 自测练习答案

- 18.1 a) 输入重定向 (<)。b) 输出重定向 (>)。c) 输出添加 (>>)。d) 管道 (|)。e) 省略号 (... )。f) `va_start`。g) `va_arg`。h) `va_end`。i) `argc`。j) `argv`。k) `make`、`makefile`。l) `exit`。m) `atexit`。n) `const`。o) 后缀。p) `signal`。q) `raise`。r) `calloc`。s) `realloc`。t) 联合体。u) `union`。

## 练习

- 18.2 编写一个程序, 计算传递给具有变长参数表函数 `product` 的一系列整数的积。用不同个数的参数测试所编写的函数。
- 18.3 编写一个程序, 打印出命令行参数。
- 18.4 编写一个程序, 按升序或降序存储一个整数数组。用命令行参数传递表示按升序存储的参数 `-a` 或按降序存储的参数 `-d`。注: 这是在 UNIX 中给程序传递选项的标准格式。
- 18.5 阅读自己系统的手册, 了解信号处理函数库 (`signal.h`) 所支持的信号。编写一个程序, 程序中包含了处理标准信号 `SIGABRT` 和 `SIGINT` 的函数 (调用函数 `abort` 来产生信号 `SIGABRT`, 键入 `<ctrl> c` 产生信号 `SIGINT`)。

- 18.6 编写一个动态分配整数数组的程序。先从键盘输入数组的大小和元素值，打印出数组的值。然后，重新分配该数组的大小，使其元素个数减少一半，打印出留在数组中的元素值，看看这些值是否是和原始数组前一半的元素值相同。
- 18.7 编写一个程序，该程序有两个都是文件名的命令行参数。程序从第一个文件中逐个读取字符，然后以相反的顺序把这些字符写入第二个文件中。
- 18.8 编写一个程序，用 `goto` 语句模拟一个打印如下星号正方形的嵌套式循环结构。

```
* * * * *
*       *
*       *
*       *
* * * * *
```

该程序只能使用下列三条输出语句：

```
cout << '*';
cout << ' ';
cout << endl;
```

- 18.9 定义联合体 `data`，包含 `char c`、`short s`、`long l`、`float f` 和 `double d`。
- 18.10 建立一个联合体 `union integer`，其成员包括 `char c`、`short s`、`int i` 和 `long l`。编写一个程序，读取 `char`、`short`、`int` 和 `long` 类型的值，把他们存入类型为 `union Integer` 的变量中。用对应的类型（`char`、`short`、`int` 和 `long` 类型）分别打印出联合体的每一个变量。打印出的值都正确吗？
- 18.11 建立一个联合体 `union FloatingPoint`，其成员包括 `float f`、`double d` 和 `long double l`。编写一个程序，读取 `float`、`double` 和 `long double` 类型的值，把他们存入类型为 `union FloatingPoint` 的联合体变量中。用对应的类型（`float`、`double` 和 `long double` 类型）分别打印出联合体的每一个变量。打印出的值都正确吗？
- 18.12 给出联合体的定义：

```
union A {
    float y;
    char *z;
};
```

下列语句中哪一条能正确地初始化这个联合体？

- a) `A p = B; // B is of same type as A`
- b) `A q = x; // x is a float`
- c) `A r = 3.14159;`
- d) `A s = { 79.63 };`
- e) `A t = { "Hi There!" };`
- f) `A u = { 3.14159, "Pi" };`

## 第 19 章 string 类与字符串流处理

### 教学目标

- 用 C++ 标准库中的 string 类将 string 作为成熟对象
- 赋值、连接、比较、查找与交换 string
- 确定 string 的特性
- 寻找、替换与插入 string 中的字符
- 将 string 变为 C 语言式字符串
- 使用 string 迭代器
- 在内存中进行 string 的输入与输出

### 19.1 简介<sup>①</sup>

C++ 模板类 `basic_string` 提供了复制、查找等典型字符串操作。模板定义和所有支持功能均已在 `namespace std` 中定义，定义中包括下列 `typedef` 语句：

```
typedef basic_string< char > string;
```

对 `basic_string< char >` 生成别名类型 `string`。对 `wchar_t` 类型也提供了 `typedef` 语句。`wchar_t` 类型存放两个字节（16 位）字符，支持外国语言。本章只使用 `string`，只有包括 C++ 标准库头文件 `<string>` 才能使用 `string`（注意：类型 `wchar_t` 常用于表示 Unicode（16 位字符），但标准中没有固定 `wchar_t` 的长度）。

`string` 对象可以用构造函数参数初始化，如下所示：

```
string s1( "Hello" ); // creates string from const char *
```

生成包含 "Hello" 字符的字符串，但可能不含终止符 '\0'，也可以用两个构造函数参数初始化，如下所示：

```
string s2( 8, 'x' ); // string of 8 'x' characters
```

生成包含 8 个 'x' 字符的 `string`。`string` 类还提供默认构造函数和复制构造函数。

`string` 也可以通过 `string` 定义中的替换构造语法初始化，如下所示：

```
string month = "March";
```

记住，上述声明中的运算符 (=) 不是赋值，而是调用 `string` 类的复制构造函数，隐式进行转换。

注意 `string` 类没有在 `string` 定义中提供 `int` 或 `char` 向 `string` 的转换。例如，下列定义：

```
string error1 = 'c';
```

---

① 本章是与作者在 Deitel & Associates 公司的同事 Tem Nieto 合写的。

```
string error2( 'u' );
string error3 = 22;
string error4( 8 );
```

会造成语法错误。注意可以在赋值语句中将一个字符赋给 string 对象，例如：

```
s = 'n';
```

#### 常见编程错误 19.1

试图在声明中通过赋值或通过构造函数参数把 int 或 char 转换为 string 是个语法错误。

#### 常见编程错误 19.2

构造的 string 太长而无法表达时会抛出 length\_error 异常。

与 C 语言式 char \* 字符串不同的是，string 不一定要用 null ( '\0' ) 终止。string 的长度存放在 string 对象中，可以用成员函数 length 读取。下标运算符 [ ] 可以用于访问 string 中的各个字符。和 C 语言式字符串一样，string 的第一个下标为 0，最后一个下标为 length-1。注意字符串不是指针，如果 s 是 string，则表达式 &s[ 0 ] 不等于 s。

大多数 string 成员函数取开始下标地址和操作的字符数为参数。

调用 string 函数时，如果传入 string 成员函数的值大于要处理的 string 字符数长度，则该值设置为 string 余下部分的长度。例如，如果函数操作长度为 50 的 string，传递开始下标 2 和字符数 100 时，则字符数为 48 ( 50-2 )。

重载流读取运算符 ( >> ) 以支持 string 的操作。下列语句：

```
string stringObject;
cin >> stringObject;
```

从标准输入设备读取 string。输入用空白字符分开。函数 getline ( 在头文件 <string> 中 ) 也对 string 重载。下列语句：

```
string s;
getline( cin, s );
```

从键盘读取 string 到 s 中。输入用换行符 ( '\n' ) 分开。

## 19.2 string 的赋值与连接

图 19.1 的程序演示了 string 的赋值与连接。

```
1 // Fig. 19.1: fig19_01.cpp
2 // Demonstrating string assignment and concatenation
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s1( "cat" ), s2, s3;
10
11     s2 = s1;           // assign s1 to s2 with =
```



```

12  s3.assign( s1 ); // assign s1 to s3 with assign()
13  cout << "s1: " << s1 << "\ns2: " << s2 << "\ns3: "
14      << s3 << "\n\n";
15
16  // modify s2 and s3
17  s2[ 0 ] = s3[ 2 ] = 'r';
18
19  cout << "After modification of s2 and s3:\n"
20      << "s1: " << s1 << "\ns2: " << s2 << "\ns3: ";
21
22  // demonstrating member function at()
23  int len = s3.length();
24  for ( int x = 0; x < len; ++x )
25      cout << s3.at( x );
26
27  // concatenation
28  string s4( s1 + "apult" ), s5; // declare s4 and s5
29
30  // overloaded +=
31  s3 += "pet"; // create "carpet"
32  s1.append( "acomb" ); // create "catacomb"
33
34  // append subscript locations 4 thru the end of s1 to
35  // create the string "comb" (s5 was initially empty)
36  s5.append( s1, 4, s1.size() );
37
38  cout << "\n\nAfter concatcenation:\n" << "s1: " << s1
39      << "\ns2: " << s2 << "\ns3: " << s3 << "\ns4: " << s4
40      << "\ns5: " << s5 << endl;
41
42  return 0;
43 }

```

**输出结果:**

```

s1: cat
s2: cat
s3: cat

```

After modification of s2 and s3:

```

s1: cat
s2: rat
s3: car

```

After concatenation:

```

s1: catacomb
s2: rat
s3: carpet
s4: catapult
s5: comb

```

图 19.1 string 的赋值与连接

第4行包括 string 类的 string 头文件。三个 string s1、s2 和 s3 在第9行生成。第11行:

```

s2 = s1; // assign s1 to s2 with =

```

将 `string s1` 赋值到 `s2`。发生赋值之后，`s2` 是 `s1` 的副本，但 `s2` 与 `s1` 并没有任何联系。第 12 行：

```
s3.assign( s1 );           // assign s1 to s3 with assign()
```

用成员函数 `assign` 将 `s1` 复制到 `s3`，建立另一个副本（即 `s1` 和 `s3` 是独立的对象）。`string` 类还提供了 `assign` 函数的重载版本，复制指定的字符串，例如：

```
myString.assign( s, start, numberOfChars );
```

其中 `s` 是要复制的 `string`，`start` 是开始下标，`numberOfChars` 是要复制的字符数。

第 17 行：

```
s2[ 0 ] = s3[ 2 ] = 'r';
```

用下标运算符将 `'r'` 赋给 `s3[ 2 ]`（形成 `"car"`），将 `'r'` 赋给 `s2[ 0 ]`（形成 `"rat"`），然后输出 `string`。

第 23 行到 25 行：

```
int len = s3.length();
for ( int x = 0; x < len; ++x )
    cout << s3.at( x );
```

通过 `for` 循环用 `at` 函数一次一个字符地输出 `s3` 内容。函数 `at` 提供检查访问或范围检查，即越过 `string` 边界时会抛出 `out_of_range` 异常（见第 13 章对异常处理的介绍）。注意下标运算符 `[]` 不提供检查访问。这与数组中的用法一致。

#### 常见编程错误 19.3

用 `at` 函数访问 `string` 时，越过 `string` 边界时会抛出 `out_of_range` 异常。

#### 常见编程错误 19.4

用下标运算符访问 `string` 长度以外的元素是个逻辑错误。

第 28 行声明字符串 `s4` 并用重载的加法运算符 `+` 初始化为 `s1` 与 `"apult"` 连接的结果，重载的加法运算符在 `string` 类中表示连接。第 31 行：

```
s3 += "pet";               // create "carpet"
```

用加号赋值运算符 `+=` 连接 `s3` 与 `"pet"`。

第 32 行：

```
s1.append( "acomb" );      // create "catacomb"
```

用函数 `append` 连接 `s1` 与 `"acomb"`。第 36 行：

```
s5.append( s1, 4, s1.size() );
```

将 `s1` 中的字符添加到 `s5` 中。`s1` 中第 4 个字符到末尾的字符连接到 `s5` 中。函数 `size` 返回 `string s1` 中的字符数。

## 19.3 比较 string

`string` 类提供了比较 `string` 的函数。图 19.2 的程序演示了 `string` 的比较功能。

```
1 // Fig. 19.2: fig19_02.cpp
2 // Demonstrating string comparison capabilities
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s1( "Testing the comparison functions." ),
10            s2("Hello" ), s3( "stinger" ), z1( s2 );
11
12     cout << "s1: " << s1 << "\ns2: " << s2
13          << "\ns3: " << s3 << "\nz1: " << z1 << "\n\n";
14
15     // comparing s1 and z1
16     if ( s1 == z1 )
17         cout << "s1 == z1\n";
18     else { // s1 != z1
19         if ( s1 > z1 )
20             cout << "s1 > z1\n";
21         else // s1 < z1
22             cout << "s1 < z1\n";
23     }
24
25     // comparing s1 and s2
26     int f = s1.compare( s2 );
27
28     if ( f == 0 )
29         cout << "s1.compare( s2 ) == 0\n";
30     else if ( f > 0 )
31         cout << "s1.compare( s2 ) > 0\n";
32     else // f < 0
33         cout << "s1.compare( s2 ) < 0\n";
34
35     // comparing s1 (elements 2 - 5) and s3 (elements 0 - 5)
36     f = s1.compare( 2, 5, s3, 0, 5 );
37
38     if ( f == 0 )
39         cout << "s1.compare( 2, 5, s3, 0, 5 ) == 0\n";
40     else if ( f > 0 )
41         cout << "s1.compare( 2, 5, s3, 0, 5 ) > 0\n";
42     else // f < 0
43         cout << "s1.compare( 2, 5, s3, 0, 5 ) < 0\n";
44
45     // comparing s2 and z1
46     f = z1.compare( 0, s2.size(), s2 );
47
48     if ( f == 0 )
49         cout << "z1.compare( 0, s2.size(), s2 ) == 0" << endl;
50     else if ( f > 0 )
51         cout << "z1.compare( 0, s2.size(), s2 ) > 0" << endl;
52     else // f < 0
53         cout << "z1.compare( 0, s2.size(), s2 ) < 0" << endl;
54
55     return 0;
```

```
56 }
```

**输出结果:**

```
s1: Testing the comparison functions.
s2: Hello
s3: stinger
z1: Hello

s1 > z1
s1.compare( s2 ) > 0
s1.compare( 2, 5, s3, 0, 5 ) == 0
z1.compare( 0, s2.size(), s2 ) == 0
```

图 19.2 比较 string

程序在第 9 行和第 10 行声明 4 个 string:

```
string s1( "Testing the comparison functions." ),
        s2( "Hello" ), s3( "stinger" ), z1( s2 );
```

并在第 12 行和第 13 行输出每个 string。第 16 行的下列条件:

```
s1 == z1
```

测试 s1 与 z1 的相等性。如果条件为 true, 则输出 "s1 == z1"。如果条件为 false, 则测试第 19 行的下列条件:

```
s1 > z1
```

这里演示和未演示的所有重载的运算符函数 (!=、<、>= 和 <=) 都返回 bool 值。

第 26 行:

```
int f = s1.compare( s2 );
```

用 string 函数 compare 测试 string s2 与 s1。声明一个变量 f, 如果两个字符串相等, 则赋值为 0; 如果 s1 在词法上 (lexicographically) 大于 s2, 则赋值为正值; 如果 s1 在词法上小于 s2, 则赋值为负值。

第 36 行:

```
f = s1.compare( 2, 5, s3, 0, 5 );
```

用函数 compare 的重载版本比较 s1 与 s3 的局部。前两个参数 (2 和 5) 指定 s1 中与 s3 比较部分的开始下标和长度。第三个参数是与 s1 比较的 string。最后两个参数 (0 和 5) 是与 s1 比较的 string 的开始下标和长度。声明一个变量 f, 如果两个字符串相等, 则赋值为 0; 如果 s1 在词法上大于 s3, 则赋值为正值; 如果 s1 在词法上小于 s3, 则赋值为负值。然后根据 f 的值打印一个 string。

第 46 行:

```
f = z1.compare( 0, s2.size(), s2 );
```

用另一个重载的 compare 函数比较 z1 与 s2。第一个参数指定比较中 z1 的开始下标, 第二个参数指定比较中 z1 的长度。函数 size 返回指定 string 中的字符数。最后一个参数是与 z1 比较的 string。声

明一个变量 `f`，如果两个字符串相等，则赋值为 0；如果 `z1` 在词法上大于 `s2`，则赋值为正值；如果 `z1` 在词法上小于 `s2`，则赋值为负值。然后根据 `f` 的值打印一个 `string`。

## 19.4 子串

`string` 类提供 `substr` 函数，可以读取 `string` 的子串。图 19.3 的程序演示了 `substr`。

程序在第 9 行声明和初始化 `string`。下列语句：

```
cout << s.substr( 7, 5 ) << endl;
```

用函数 `substr` 从 `s` 读取一个子串。第一个参数指定子串的开始下标，第二个参数指定读取的字符数。

```
1 // Fig. 19.3: fig19_03.cpp
2 // Demonstrating function substr
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s( "The airplane flew away." );
10
11     // retrieve the substring "plane" which
12     // begins at subscript 7 and consists of 5 elements
13     cout << s.substr( 7, 5 ) << endl;
14
15     return 0;
16 }
```

**输出结果：**

Plane

图 19.3 用 `substr` 读取 `string` 的子串

## 19.5 交换 `string`

`string` 类提供 `swap` 函数，可以交换 `string`。图 19.4 的程序演示交换两个 `string`。

第 9 行声明和初始化 `string`： `first` 和 `second`。然后输出每个 `string`。第 13 行：

```
first.swap( second );
```

用函数交换 `first` 和 `second` 的值。然后再次输出两个 `string`，确认其实际已经交换。

```
1 // Fig. 19.4: fig19_04.cpp
2 // Using the swap function to swap two strings
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
```

```
9   string first( "one" ), second( "two" );
10
11   cout << "Before swap:\n first: " << first
12       << "\nsecond: " << second;
13   first.swap( second );
14   cout << "\n\nAfter swap:\n first: " << first
15       << "\nsecond: " << second << endl;
16
17   return 0;
18 }
```

**输出结果:**

```
Before swap:
  first: one
second: two
```

```
After swap:
  first: one
second: one
```

图 19.4 用 swap 交换 string

## 19.6 string 的特性

string 类提供了计算 string 大小、长度、容量、最大长度和其他特性的函数。string 的大小或长度 (size 或 length) 是 string 中目前存放的字符数。string 的容量 (capacity) 是 string 中不必增加内存即可存放的总的元素个数。最大长度 (maximum size) 是 string 对象中可存放 string 的最大长度。图 19.5 的程序演示了获得 string 大小、长度和其他特性的 string 类函数。

```
1 // Fig. 19.5: fig19_05.cpp
2 // Demonstrating functions related to size and capacity
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 void printStats( const string & );
8
9 int main()
10 {
11     string s;
12
13     cout << "Stats before input:\n";
14     printStats( s );
15
16     cout << "\n\nEnter a string: ";
17     cin >> s; // delimited by whitespace
18     cout << "The string entered was: " << s;
19
20     cout << "\nStats after input:\n";
21     printStats( s );
22
23     s.resize( s.length() + 10 );
```

```
24     cout << "\n\nStats after resizing by (length + 10):\n";
25     printStats( s );
26
27     cout << endl;
28     return 0;
29 }
30
31 void printStats( const string &str )
32 {
33     cout << "capacity: " << str.capacity()
34          << "\nmax size: " << str.max_size()
35          << "\nsize: " << str.size()
36          << "\nlength: " << str.length()
37          << "\nempty: " << ( str.empty() ? "true": "false" );
38 }
```

**输出结果:**

```
Stats before input:
Capacity: 0
Max size: 4294967293
size: 0
length: 0
empty: true
```

```
Enter a string: tomato soup
The string entered was: tomato
Stats after input:
capacity: 31
max size: 4294967293
size: 6
length: 6
empty: false
```

```
Stats after resizing by (length + 10):
capacity: 31
max size: 4294967293
size: 16
length: 16
empty: false
```

图 19.5 打印 string 的特性

程序在第 11 行声明空的 string s，并将其传入函数 printStats（第 14 行）。空 string 是不包含任何字符的 string。字符串 "tomato" 从键盘输入。注意 string 用空白字符分隔，以免输入剩余的字符串。

函数 printStats 取 const string 的引用为参数，输出 string 的容量（用 capacity 函数）、最大长度（用 max\_size 函数）、大小（用 size 函数）、长度（用 length 函数）和是否为空字符串（用 empty 函数）。最初调用 printStats 时，表示 s 的初始容量、大小和长度为 0。由于容量为 0，因此在 s 中存放字符时，需要增加内存。大小与长度为 0 表示 s 中当前没有存放字符。大小与长度总是相同的。目前版本中最大长度是 4294967293。字符串 s 是个空字符串，因此函数 empty 返回 true。

第 17 行将一个 string 输入 s 中。注意这里使用流读取运算符 >>。第 23 行：

```
s.resize( s.length() + 10 );
```

用函数 `resize` 将 `s` 的长度增加 10 个字符。

## 19.7 寻找 string 中的字符

`string` 类提供了寻找 `string` 中的字符与字符串的函数。图 19.6 的程序演示了寻找函数。所有寻找函数都是 `const` 类型。

```

1 // Fig. 19.6: fig19_06.cpp
2 // Demonstrating the string find functions
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     // compiler concatenates all parts into one string literal
10    string s( "The values in any left subtree"
11              "\nare less than the value in the"
12              "\nparent node and the values in"
13              "\nany right subtree are greater"
14              "\nthan the value in the parent node" );
15
16    // find "subtree" at locations 23 and 102
17    cout << "Original string:\n" << s
18          << "\n\n(find) \"subtree\" was found at: "
19          << s.find( "subtree" )
20          << "\n\n(rfind) \"subtree\" was found at: "
21          << s.rfind( "subtree" );
22
23    // find 'p' in parent at locations 62 and 144
24    cout << "\n\n(find_first_of) character from \"qpxz\" at: "
25          << s.find_first_of( "qpxz" )
26          << "\n\n(find_last_of) character from \"qpxz\" at: "
27          << s.find_last_of( "qpxz" );
28
29    // find 'b' at location 25
30    cout << "\n\n(find_first_not_of) first character not\n"
31          << "    contained in \"heTv lusi nodrpayft\": "
32          << s.find_first_not_of( "heTv lusi nodrpayft" );
33
34    // find '\n' at location 121
35    cout << "\n\n(find_last_not_of) first character not\n"
36          << "    contained in \"heTv lusi nodrpayft\": "
37          << s.find_last_not_of( "heTv lusi nodrpayft" ) << endl;
38
39    return 0;
40 }

```

**输出结果：**

```

Original string:
The values in any left subtree
are less than the value in the

```



```

parent node and the values in
any right subtree are greater
than the value in the parent node

(find) "subtree" was found at: 23
(rfind) "subtree" was found at: 102
(find_first_of) character from "qpxz" at: 62
(find_last_of) character from "qpxz" at: 144
(find_first_not_of) first character not
    contained in "heTv lusinodrpayft": 25
(find_last_not_of) first character not
    contained in "heTv lusinodrpayft": 121

```

图 19.6 演示 string find 函数的程序

第10行声明和初始化字符串s。编译器将五个字符串连接为一个字符串。为了避免语法错误，每个字符串的末尾应当放在双引号中，然后再转入下一行开始另一个字符串。

#### 常见编程错误 19.5

不用双引号终止字符串是个语法错误。

第19行的插入操作：

```
s.find( "subtree" )
```

试图用 find 函数寻找 string s 中的 string "subtree"。如果找到这个 string，则返回该 string 的开始位置下标。如果找不到这个 string，则返回该 string::npos 值（string 类中定义的 public static 常量）。这个值由 string find 相关函数返回，表示子串或字符不在 string 中。

第21行用流插入输出最后一项：

```
s.rfind( "subtree")
```

使用 rfind 函数从后向前查找 string s。如果找到这个 string，则返回该 string 的开始位置下标。如果找不到这个 string，则返回 string::npos 值。注意：除非另有说明，本节的其他寻找函数返回相同的数值类型。另外，常量 string::npos 用于不同上下文中表示 string 的所有元素。

第25行的下列语句：

```
s.find_first_of( "qpxz" )
```

用函数 find\_first\_of 寻找 string s 中第一次出现的 "qpxz" 等字符。搜索从 s 开头开始，在第62位找到字符 'p'。

第27行的下列语句：

```
s.find_last_of( "qpxz" )
```

用函数 find\_last\_of 寻找 string s 中最后一次出现的 "qpxz" 等字符。搜索从 s 末尾开始，在第144位找到字符 'p'。

第32行的下列语句：

```
s.find_first_not_of( "heTv lusinodrpayft" );
```

用 find\_first\_not\_of 函数寻找 string s 中第一个不在 "heTv lusinodrpayft" 中的字符。搜索从 s 开头开始。

第37行的下列语句:

```
s.find_last_not_of( "heTv lusinodrpayft" );
```

用 `find_last_not_of` 函数寻找 string `s` 中最后一个不在 "heTv lusinodrpayft" 中的字符。搜索从 `s` 末尾开始。

## 19.8 替换 string 中的字符

图 19.7 的程序演示替换与清除字符的 `string` 函数。

```
1 // Fig. 19.7: fig19_07.cpp
2 // Demonstrating functions erase and replace
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     // compiler concatenates all parts into one string
10    string s( "The values in any left subtree"
11             "\nare less than the value in the"
12             "\nparent node and the values in"
13             "\nany right subtree are greater"
14             "\nthan the value in the parent node" );
15
16    // remove all characters from location 62
17    // through the end of s
18    s.erase( 62 );
19
20    // output the new string
21    cout << "Original string after erase:\n" << s
22         << "\n\nAfter first replacement:\n";
23
24    // replace all spaces with a period
25    int x = s.find( " " );
26    while ( x < string::npos ) {
27        s.replace( x, 1, "." );
28        x = s.find( " ", x + 1 );
29    }
30
31    cout << s << "\n\nAfter second replacement:\n";
32
33    // replace all periods with two semicolons
34    // NOTE: this will overwrite characters
35    x = s.find( "." );
36    while ( x < string::npos ) {
37        s.replace( x, 2, "xxxxx;yyy", 5, 2 );
38        x = s.find( ".", x + 1 );
39    }
40
41    cout << s << endl;
42    return 0;
43 }
```

**输出结果:**

```
Original string after erase:
The values in any left subtree
are less than the value in the
After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the

After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

图 19.7 演示清除与替换 string 字符的函数

程序声明和初始化 string s。第 18 行:

```
s.erase( 62 );
```

用函数 erase 清除 s 中从元素 62 开始到末尾的所有字符。

第 25 行到第 29 行:

```
x = x.find( " " );
while ( x < string::npos ) {
    s.replace( x, 1, "." );
    x = s.find( " ", x + 1 );
}
```

用函数 find 寻找每个空格符。然后调用 replace 函数将每个空格符换成圆点。函数 replace 取三个参数: 开始下标、要替换的字符数和替换字符串。常量 string::npos 表示最大字符串长度。函数 find 在到达 s 结尾时返回 string::npos。

第 35 行到第 39 行:

```
x = s.find( "." );
while ( x < string::npos ) {
    s.replace( x, 2, "xxxxx;yyy", 5, 2 );
    x = s.find( ".", x + 1 );
}
```

用 find 函数寻找每个圆点, 并用 replace 函数将每个圆点及其后面的字符变成两个分号。传入 replace 函数的参数是替换操作的开始下标、要替换的字符数、替换字符串 (其子串用来替换字符) 以及替换字符串中替换子串的开始位置下标和字符数。

## 19.9 在 string 中插入字符

string 类提供了在 string 中插入字符的函数。图 19.8 的程序演示了 string insert 功能。

```
1 // Fig. 19.8: fig19_08.cpp
2 // Demonstrating the string insert functions.
3 #include <iostream>
```

```
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s1( "beginning end" ),
10             s2( "middle " ), s3( "12345678" ), s4( "xx" );
11
12     cout << "Initial strings:\ns1: " << s1
13          << "\ns2: " << s2 << "\ns3: " << s3
14          << "\ns4: " << s4 << "\n\n";
15
16     // insert "middle" at location 10
17     s1.insert( 10, s2 );
18
19     // insert "xx" at location 3 in s3
20     s3.insert( 3, s4, 0, string::npos );
21
22     cout << "Strings after insert:\ns1: " << s1
23          << "\ns2: " << s2 << "\ns3: " << s3
24          << "\ns4: " << s4 << endl;
25
26     return 0;
27 }
```

**输出结果:**

```
Initial strings:
s1: beginning end
s2: middle
s3: 12345678
s4: xx

Strings after insert:
s1: beginning middle end
s2: middle
s3: 123xx45678
s4: xx
```

图 19.8 演示 string insert 功能

程序声明和初始化 4 个 string: s1、s2、s3 和 s4。然后输出每个 string。第 17 行:

```
s1.insert( 10, s2 );
```

用函数 insert 在元素 10 之前插入 string s2。第 20 行:

```
s3.insert( 3, s4, 0, string::npos );
```

用 insert 在 s3 的第 3 个元素的后面插入 s4。最后两个参数指定 s4 中要插入的开始位置下标和字符数。

**性能提示 19.1**

插入操作可能造成额外的内存管理操作,从而使性能下降。

## 19.10 转换成C语言式char\*字符串

string类提供了将string转换成C语言式char\*字符串的函数。前面曾介绍过,与C语言式char\*字符串不同的是,string不需要null终止。这种转换函数可以让指定的函数取C语言式char\*字符串作为参数。图19.9演示了将string转换成C语言式字符串的程序。

```
1 // Fig. 19.9: fig19_09.cpp
2 // Converting to C-style strings.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s( "STRINGS" );
10    const char *ptr1 = 0;
11    int len = s.length();
12    char *ptr2 = new char[ len + 1 ]; // including null
13
14    // Assign to pointer ptr1 the const char * returned by
15    // function data(). NOTE: this is a potentially dangerous
16    // assignment. If the string is modified, the pointer
17    // ptr1 can become invalid.
18    ptr1 = s.data();
19
20    // copy characters out of string into allocated memory
21    s.copy( ptr2, len, 0 );
22    ptr2[ len ] = 0; // add null terminator
23
24    // output
25    cout << "string s is " << s
26          << "\ns converted to a C-Style string is "
27          << s.c_str() << "\nptr1 is ";
28
29    for ( int k = 0; k < len; ++k )
30        cout << *( ptr1 + k ); // use pointer arithmetic
31
32    cout << "\nptr2 is " << ptr2 << endl;
33    delete [] ptr2;
34    return 0;
35 }
```

**输出结果:**

```
string s is STRINGS
s converted to a C-Style string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS
```

图19.9 将string转换成C语言式字符串和字符数组

程序声明一个string类型的变量、一个int类型的变量和两个指针变量。string s初始化为"STRINGS", ptr1初始化为0, len初始化为s的长度。动态分配内存并连接指针 ptr2。

第 18 行:

```
ptr1 = s.data();
```

将 `data` 返回的非 `null` 终止 C 语言式字符数组 (`const char *` 类型) 赋给 `ptr1`。注意例子中没有修改 `string s`。如果要修改 `s`, 则 `ptr1` 可能变成无效, 从而造成无法预料的结果。

第 21 行:

```
s.copy( ptr2, len, 0 );
```

用函数 `copy` 将 `s` 复制到 `ptr2` 所指的数组中。从 `string` 向 C 语言式字符串的转换是隐式的。第 22 行在数组 `ptr2` 中加上一个 `null` 终止符。

第 27 行的第一个流插入语句:

```
<< s.c_str()
```

表示转换 `string s` 时从 `c_str` 返回 `null` 终止的 `const char *` 类型的字符串。注意 `data` 返回的字符数组和 `c_str` 返回的 C 语言式字符串的存在时间有限, 而且属于 `class string`, 不能用 `delete` 删除。

第 29 行和 30 行用指针算法输出 `ptr1` 所指的数组。在第 32 行和第 33 行, 输出 `ptr2` 所指的 C 语言式字符串并删除为 `ptr2` 分配的内存, 避免造成内存泄漏。

#### 常见编程错误 19.6

`data` 或 `copy` 返回的字符数组不用空 (`null`) 字符终止可能造成执行时错误。

#### 编程技巧 19.1

如果可能, 尽量使用更健壮的 `string` 而不用 C 语言式字符串。

#### 常见编程错误 19.7

将包含一个或几个空字符的 `string` 变为 C 语言式字符串可能造成逻辑错误。空字符是 C 语言式字符串的终止符。

## 19.11 迭代器

`string` 类提供了向前和向后遍历 `string` 的迭代器 (iterator)。迭代器提供了访问各个字符的语法, 类似于指针操作。迭代器不检查范围。注意, 本节提供了演示迭代器用法的简单例子, 下一章将介绍迭代器更健壮的法。图 19.10 的程序演示了迭代器。

```
1 // Fig. 19.10: fig19_10.cpp
2 // Using an iterator to output a string.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s( "Testing iterators" );
10    string::const_iterator i1 = s.begin();
11
12    cout << "s = " << s
13         << "\n(Using iterator i1) s is: ";
```

```

14
15     while ( il != s.end() ) {
16         cout << *il;    // dereference iterator to get char
17         ++il;           // advance iterator to next char
18     }
19
20     cout << endl;
21     return 0;
22 }

```

**输出结果：**

```

s = Testing iterators
(Using iterator il) s is: Testing iterators

```

图 19.10 用迭代器输出 string

第 9 行和第 10 行：

```

string s( "Testing iterators" );
string::const_iterator il = s.begin();

```

声明 string s 和 string::const\_iterator il。const\_iterator 是个迭代器，不能修改所迭代的容器（这里是字符串）。迭代器 il 初始化为 s 的开头，这是用 string 类函数 begin 指定的。begin 有两种版本，一种返回通过非 const string 迭代的 iterator，而 const 版本返回通过 const string 迭代的 const\_iterator。然后输出字符串 s。

第 15 行到第 18 行：

```

while ( il != s.end() ) {
    cout << *il;
    ++il;
}

```

用迭代器 il 遍历 s。函数 end 返回 s 最后一个元素后面第一个位置的迭代器。每个位置的内容首先通过复引用迭代器而打印，就像指针复引用一样，然后迭代器用 ++ 运算符向前移一位。

string 类提供的成员函数 rend 和 rbegin 可以从 string 末尾向 string 开头逆向访问 string 的各个字符。成员函数 rend 和 rbegin 可以返回 reverse\_iterators 和 const\_reverse\_iterators（根据 string 为 const 或非 const）。我们将要求读者在练习中演示这个问题。第 20 章将更详细地介绍迭代器和逆向迭代器。

#### 测试与调试提示 19.1

如果要进行范围检查，用 string 成员函数 at（而不用迭代器）。

## 19.12 字符串流处理

除了标准流 I/O 和文件流 I/O 外，C++ 的流 I/O 还可以从内存中的 string 输入和输出。这些功能通常称为内存中 I/O 或字符串流处理。

istringstream 类支持从 string 输入，ostringstream 类支持从 string 输出。类名 istringstream 和 ostringstream 实际上都是别名，这些名称用 typedef 定义：

```

typedef basic_istringstream< char > istringstream;

```

```
typedef basic_ostringstream< char > ostringstream;
```

类 `basic_istringstream` 和 `basic_ostringstream` 提供与 `istream` 和 `ostream` 相同的功能，而且又有其他指定内存中格式化的成员函数。使用内存中格式化的程序应包括 `<sstream>` 和 `<iostream>` 头文件。

这些方法的一个应用是数据验证。程序可以一次一整行地从输入流读取到 `string` 中，接着验证程序可以检查 `string` 内容，并在需要时纠正或修复数据。然后程序可以继续从 `string` 输入，这时输入数据已经得到正确的格式。

`string` 输出也可以很好地利用 C++ 流强大的输出格式化功能。数据可以在 `string` 中准备，模拟编辑的屏幕格式。该 `string` 可以写入磁盘文件，保存屏幕映像。

`ostringstream` 对象用 `string` 对象保存输出的数据。`ostringstream` 成员函数 `str` 返回内存中对 `string` 的 `string` 引用。

图 19.11 演示了 `ostringstream` 对象。程序生成 `ostringstream` 对象 `outputString`（第 10 行）并用流插入运算符向该对象输出一系列 `string` 和数字值。

```
1 // Fig. 19.11: fig19_11.cpp
2 // Using a dynamically allocated ostringstream object.
3 #include <iostream>
4 #include <string>
5 #include <sstream>
6 using namespace std;
7
8 main()
9 {
10     ostringstream outputString;
11     string s1( "Output of several data types " ),
12           s2( "to an ostringstream object:" ),
13           s3( "\n          double: " ),
14           s4( "\n          int: " ),
15           s5( "\naddress of int: " );
16     double d = 123.4567;
17     int i = 22;
18
19     outputString << s1 << s2 << s3 << d << s4 << i << s5 << &i;
20     cout << "outputString contains:\n" << outputString.str();
21
22     outputString << "\nmore characters added";
23     cout << "\n\nafter additional stream insertions,\n"
24           << "outputString contains:\n" << outputString.str()
25           << endl;
26
27     return 0;
28 }
```

#### 输出结果：

```
outputString contains
output of several data types to an ostringstream object:
          double: 123.457
          int: 22
address of int: 0068FD0C

after additional stream insertions,
outputString contains:
```



```

output of several data types to an ostringstream object:
    double: 123.457
    int: 22
address of int: 0068FD0C
more characters added

```

图 19.11 使用动态分配的 ostringstream 对象

第 19 行:

```
outputString << s1 << s2 << s3 << d << s4 << i << s5 << &i;
```

输出 string s1、string s2、string s3、double d、string s4、int i、string s5 和 int i 的地址，全部输出到内存中的 outputString 中。第 20 行:

```
cout << "outputString contains:\n" << outputString.str();
```

调用 outputString.str() 输出第 19 行生成的 string。第 22 行演示了可以向 outputString 发出另一个流插入操作，向内存中的 string 添加更多数据。第 24 行在添加更多数据之后输出 string outputString。

istringstream 对象从内存中的 string 向程序变量输入数据。数据以字符形式存放在 istringstream 对象中。从 istringstream 对象输入与从任何文件输入或从标准输入设备输入时是一样的。istringstream 将 string 的结尾解释为文件尾。

图 19.12 演示了从 istringstream 对象输入。第 10 行和第 11 行:

```

string input( "Input test 123 4.7 A" );
istringstream inputString( input );

```

生成的 string input 包含数据和 istringstream 的对象 inputString，该对象用于包含 string input 中的数据。string input 包含下列数据:

```
Input test 123 4.7 A
```

当读取到程序中时，包含两个字符串 ("Input" 和 "test")、一个 int 值 (123)、一个 double 值 (4.7) 和一个 char 值 ('A')。这些数据在第 17 行分别输入到变量 string1、string2、i、d 和 c:

```
inputString >> string1 >> string2 >> i >> d >> c;
```

然后在第 19 行到第 25 行输出数据。第 30 行试图用 if/else 语句再次从 inputString 读取。由于已经没有数据，因此 if 条件求值为 false，执行 if/else 结构的 else 部分。

```

1 // Fig. 19.12: fig19_12.cpp
2 // Demonstrating input from an istringstream object.
3 #include <iostream>
4 #include <string>
5 #include <sstream>
6 using namespace std;
7
8 main()
9 {
10     string input( "Input test 123 4.7 A" );
11     istringstream inputString( input );
12     string string1, string2;
13     int i;

```

```
14 double d;
15 char c;
16
17 inputString >> string1 >> string2 >> i >> d >> c;
18
19 cout << "The following inputs were extracted\n"
20      << "from the istream object:"
21      << "\nstring: " << string1
22      << "\nstring: " << string2
23      << "\n int: " << i
24      << "\ndouble: " << d
25      << "\n char: " << c;
26
27 // attempt to read from empty stream
28 long l;
29
30 if ( inputString >> l )
31     cout << "\n\nlong value is: " << l << endl;
32 else
33     cout << "\n\ninputString is empty" << endl;
34
35 return 0;
36 }
```

#### 输出结果:

```
The following items were extracted
from the istream object
String: Input
String: test
int: 123
double: 4.7
char: A
```

```
inputString is empty
```

图 19.12 演示从 istream 对象输入

## 小结

- C++ 模板类 `basic_string` 提供了复制、查找等典型字符串操作。

- `typedef` 语句:

```
typedef basic_string< char > string;
```

对 `basic_string< char >` 生成别名类型 `string`。对 `wchar_t` 类型也提供了 `typedef` 语句。`wchar_t` 类型存放二个字节 (16 位) 字符, 支持外国语言, 但标准中没有固定 `wchar_t` 的长度。

- 只有包含 C++ 标准库头文件 `<string>` 才能使用 `string`。
- `string` 类没有在 `string` 定义中提供 `int` 或 `char` 向 `string` 的转换。
- 可以在赋值语句中将一个字符赋给 `string` 对象。
- `string` 不一定要用 `null ('\\0')` 终止。
- `string` 的长度存放在 `string` 对象中, 可以用成员函数 `length` 读取。

- 大多数 string 成员函数把开始下标地址和操作的字符数作为参数。
- 如果传入 string 成员函数的值大于要处理的 string 字符数长度, 则该值设置为 string 余下部分的长度。
- string 类提供对 string 赋值的重载的运算符 = 和成员函数 assign。
- 下标运算符[]可以用于 string 中访问各个字符。
- 函数 at 提供检查访问或范围检查, 即越过 string 边界时会抛出 out\_of\_range 异常。而下标运算符[]不提供检查访问。
- string 类提供连接 string 的重载运算符 + 与 += 和成员函数 append。
- string 类提供比较 string 的重载的运算符 ==、!=、<、>、<= 和 >=。
- string 函数 compare 比较两个 string (或子串) 时, 如果 string 相等, 则函数返回 0; 如果第一个 string 在词法上大于第二个 string, 则返回正值; 如果第一个 string 在词法上小于第二个 string, 则返回负值。
- string 类提供 substr 函数, 可以读取 string 的子串。
- string 类提供 swap 函数, 可以交换 string。
- 函数 size 和 length 返回 string 的大小或长度 (即 string 中当前存放的字符数)。
- 函数 capacity 返回 string 中不增加内存需求时可以存放的总元素个数。
- 函数 max\_size 返回 string 对象可以存放的最大元素个数。
- 函数 resize 增加 string 的长度。
- string 类寻找函数 find、rfind、find\_first\_of、find\_last\_of、find\_first\_not\_of 和 find\_last\_not\_of 寻找 string 中的字符或字符串。
- 值 string::npos 表示在要处理几个字符的函数中处理 string 的所有元素。
- 函数 erase 清除 string 的元素。
- 函数 replace 替换 string 的元素。
- 函数 insert 插入 string 的字符。
- 函数 c\_str 返回 const char\*, 指向以 null 终止的 C 语言式字符数组, 其中包含 string 的所有字符。
- string 类提供成员函数 end 和 begin, 可以访问各个 string 字符。
- string 类提供成员函数 rend 和 rbegin, 可以从后向前按相反顺序访问各个 string 字符。
- istream 类型支持从 string 输入。ostream 类型支持向 string 输出。
- 使用内核格式的程序应包括 <sstream> 和 <iostream> 头文件。
- ostream 成员函数 str 返回内存中用 ostream 对象生成的对 string 的 string 引用。

## 术语

access function    access 函数

at function    at 函数

c\_str function    c\_str 函数

capacity    容量

capacity function    capacity 函数

checked access    检查访问

compare function    compare 函数

const\_iterator

const\_reverse\_iterator

copy function    copy 函数

data function    data 函数

empty string    空字符串

equality operators: ==, !=    相等运算符

empty function    empty 函数

erase function    erase 函数

find function    find 函数

find functions 寻找函数	rbegin function rbegin 函数
find_first_of function find_first_of 函数	relational operators: >, <, >=, <= 关系运算符: >, <, >=, <=
find_last_of function find_last_of 函数	rend function rend 函数
find_first_not_of function find_first_not_of 函数	replace function replace 函数
find_last_not_of function find_last_not_of 函数	resize function resize 函数
getline function getline 函数	reverse_iterator
in-core I/O 内核 I/O	size function size 函数
in-memory I/O 内存中 I/O	<sstream> header file <sstream> 头文件
insert function insert 函数	str string-stream member function str 字符串流成员函数
istringstream class istringstream 类	string class string 类
iterator	<string> header file <string> 头文件
length of a string string 长度	subscript operator, [] 下标运算符[]
length function length 函数	substr function substr 函数
length_error exception length_error 异常	swap function swap 函数
max_size function max_size 函数	typedef basic_string<char> string statement typedef basic_string<char> string 语句
maximum size of a string string 的最大长度	wchar_t type wchar_t 类型
namespace std	wide characters 宽位字符
operators: += <<>> [] 运算符: += <<>> []	
ostream class ostream 类	
out_of_range exception out_of_range 异常	
range_error exception range_error 异常	

## 自测练习

### 19.1 填空:

- string 类应包括 \_\_\_\_\_ 头文件。
- string 类属于 \_\_\_\_\_ namespace。
- 函数 \_\_\_\_\_ 从 string 中清除字符。
- 函数 \_\_\_\_\_ 寻找一系列字符串中任何一个字符第一次出现的位置。

### 19.2 判断下列各题是否正确。如果不正确, 请说明原因。

- 连接可以用加法运算符 += 进行。
- string 中的字符从元素 0 开始。
- 赋值运算符 "=" 复制 string。
- C 语言式字符串属于 string。

### 19.3 寻找下列各题的错误并说明如何纠正。

- ```
string sv( 28 ); // construct sv
string bc( 'z' ); // construct bc
```
- ```
// assume std namespace is known
const char *ptr = name.data(); // name is "joe bob"
ptr[ 3 ] = '-';
cout << ptr << endl;
```

## 自测练习答案

- 19.1 a) string。 b) std。 c) erase。 d) find\_first\_of。
- 19.2 a) 正确。
- b) 正确。
- c) 正确。
- d) 不正确。string 是个提供许多不同服务的对象，而 C 语言式字符串不提供任何服务。C 语言式字符串是用 null ('\0') 终止的，而 string 不是。
- 19.3 a) 所传入参数的构造函数不存在。应使用其他有效构造函数，必要时将参数变为 string。
- b) data 函数不增加 null 终止符。改用 c\_str。

## 练习

- 19.4 填空：
- a) 函数 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_ 将 string 变为 C 语言式字符串。
- b) 函数 \_\_\_\_\_ 用于赋值。
- c) 函数 rbegin 的返回类型为 \_\_\_\_\_。
- d) 函数 \_\_\_\_\_ 用于读取子串。
- 19.5 判断下列各题是否正确。如果不正确，请说明原因。
- a) string 是由 null 终止的。
- b) 函数 max\_size 返回 string 的最大长度。
- c) 函数 at 能够抛出 out\_of\_range 异常。
- d) 函数 begin 返回一个 iterator。
- e) string 默认按引用传递。
- 19.6 寻找下列各题的错误并说明如何纠正。
- a) `std::cout << s.data() << std::endl; // s is "hello"`
- b) `erase( s.rfind( "x" ), 1 ); // s is "xenon"`
- c) `string& foo( void )`  
{  
    string s( "Hello" );  
    ... // other statements of function  
    return;  
}
- 19.7 (简单加密) Internet 中的有些信息可能通过 "rot13" 的简单算法进行加密，每个字母在字母表中移动 13 位。这样，a 对应于 n、x 对应于 k 等等。rot13 是个对称密钥加密算法。使用对称密钥加密算法时，加密和解密使用相同的密钥。
- a) 编写一个用 rot13 加密消息的程序。
- b) 编写一个用相同密钥解密消息的程序。
- c) 编写 a) 和 b) 之后，简述下列问题：如果不知道 b) 的密钥，用现有资源进行破译的难度如何？如果你有功能强大的计算机（如 Cray 超级计算机）呢？练习 19.27 要编写这种程序。
- 19.8 编写一个程序，用迭代器演示 rend 和 rebegin 函数的用法。
- 19.9 编写自己的 data 和 c\_str 函数版本。

- 19.10 编写一个程序, 读取几个 string, 只打印以 "r" 或 "ay" 结尾的 string。只考虑小写字母。
- 19.11 编写一个程序, 演示按引用和按值传递 string。
- 19.12 编写一个程序, 分别输入姓和名, 然后连接成一个新的 string。
- 19.13 编写一个程序, 玩吊死鬼游戏。程序选择一个单词(直接放在程序中或从一个文本文件读取), 然后显示:

```
Guess the word:  XXXXXX
```

每个 X 表示一个字母。如果猜对, 则程序显示:

```
Congratulations!!! You guessed my word. Play again? yes/no
```

输入 yes 或 no。如果猜错, 则显示身体的相应部分。

七次猜错时, 即显示吊死鬼形状如下:

```
  O
 / \|
  |
 / \|
```

每次猜测后要显示所猜内容。

- 19.14 编写一个程序, 输入 string 并逆向打印 string。将所有大写字母变成小写, 将所有小写字母变成大写。
- 19.15 编写一个程序, 用本章介绍的比较功能将一系列动物名进行排序。比较时只用大写字母。
- 19.16 编写一个程序用 string 生成一个密电。密电是将每个字母换成另一字母的消息。例如下列 string:

```
The birds name was squawk
```

可能变成:

```
xms kbypo zhqs fho obrhfu
```

注意, 空格没有加密, 这里 T 换成 x、a 换成 h 等等。大小写都一样处理。用同样方法解决练习 19.7。

- 19.17 修改上述练习, 让用户输入两个字母以破译密电。第一个字符指定密电中的字符, 第二个字符指定用户猜测的字符。例如, 如果用户输入 rg, 则用户估计 r 实际上表示 g。
- 19.18 编写一个程序, 输入句子并计算句子中的回文单词个数。回文单词就是正读反读都一样的单词。例如, "tree" 不是回文单词, 而 "noon" 是回文单词。
- 19.19 编写一个程序, 计算句子中元音字母的总数。输出每个元音字母的出现的频率。
- 19.20 编写一个程序, 在 string 中间插入字符 "\*\*\*\*\*"。
- 19.21 编写一个程序, 清除 string 中的 "by" 和 "BY" 序列。
- 19.22 编写一个程序输入一行文本, 将所有标点符号变为空格, 然后用 C 语言字符串库函数 strtok 将 string 标记化为各个单词。
- 19.23 编写一个程序并逆向打印这行文本。在程序中使用迭代器。
- 19.24 用递归方法解答练习 19.23。
- 19.25 编写一个程序, 演示使用取 iterator 参数的 erase 函数。
- 19.26 编写一个程序从 string "abcdefghijklmnopqrstuvwxyz{" 产生下列形状:

```
a
bcd
cdedc
defgfed
efghihgfe
fghijkjihgf
ghijklmlkjing
hijklmnonmlkjih
ijklmnopqponmlkji
jklmnopqrsrqponmlkj
klmnopqrstutsrqponmlk
lmnopqrstuvwutsrqponml
mnopqrstuvwxyzvwutsrqponm
nopqrstuvwxyz{zyxwutsrqpon
```

- 19.27 练习 19.7 要求编写一个简单加密算法。编写一个程序，用简单的频率替换（假设不知道密钥）解密 "rot13" 消息。加密文本中最常出现的字母换成最常用的英语字母（a、e、i、o、u、s、t、r 等）。编写文件的概率。哪个代码最容易破译？如何改进加密机制？
- 19.28 编写一个排序 string 的冒泡程序，使用 swap 函数。

## 第20章 标准模板库 (STL)

### 教学目标

- 使用模板化 STL 容器、容器适配器和“近容器”
- 用各种 STL 算法编程
- 了解算法如何用迭代器访问 STL 容器元素
- 熟悉 Internet 和万维网上的 STL 资源

### 20.1 标准模板库 (STL) 简介

除了维护性与复用性外，面向对象的最大好处是复用、复用、再复用。ANSI/ISO C++ 草案标准包括一个有许多复用组件的标准库。本章介绍标准模板库 (STL, Standard Template Library)，我们将介绍 STL 的三个主要组件：容器 (container, 常用模板化数据类型)、迭代器 (iterator) 与算法 (algorithm)。

STL 是由惠普公司的 Alexander Stepanov 和 Meng Lee 开发的，是他们在一般化编程领域研究成果的结晶，并且 David Musser 在这个领域也有重大贡献。STL 已经成为 ANSI/ISO C++ 草案标准的一部分，必将受到越来越广泛的使用。

STL 的内容相当多。我们的目的是通过本章引导学生开始有效地使用 STL。我们用三十多个“有生命力的代码”的实例程序介绍了 STL 的大部分功能，读者可以看到能够执行的 STL。

第 15 章简单介绍了数据结构。数据结构是数据的容器。在面向对象世界中，数据结构是包含对象的对象。第 8 章开发的 Array 类包含基本数据类型 int 的对象。学习第 12 章“模板”之后，就可以将 Array 类模板化成 Array<T>，从中可以实例化各种 Array 类，如 Array<int>、Array<char>、Array<double>，甚至可以实例化非基本类对象的数组，如 Array<Employee>、Array<SpaceCreature>、Array<InventoryItem>等等。

而对于链表、堆栈、队列、优先级队列等其他数据结构，如果我们要一一开发自己的版本，那么利用这些类模板编写的程序是否能与其他程序员用自己的类模板编写的程序方便地相互操作？也许很难。因此，很有必要开发一个标准化的模板化对象容器库，这就是标准模板库 (STL)。利用 STL 可以节省大量时间和精力，并得到更高质量的程序，这就是复用的好处。

#### 性能提示 20.1

对任何特定的应用程序，可能适用几种不同的 STL 容器。选择最适合的容器，使应用程序取得最佳性能（即速度与长度的平衡）。有效性是 STL 设计中的关键。

#### 性能提示 20.2

实现标准库功能可以在各种不同应用程序之间有效地操作。对有些具有独特性能要求的程序，可能需要编写自定义的实现方法。



### 软件工程视点 20.1

STL 方法可以编写常规程序,使代码不依赖于基础容器。这种编程风格称为常规化编程。

20 世纪 70 年代,我们使用的组件是控制结构和函数。20 世纪 80 年代,我们主要使用各种平台相关类库中的类。20 世纪 90 年代末推出了 STL,使组件化又上了一个台阶,成为独立于平台的类库。今后十年,这种类的数量必将呈指数形式上升。

STL 是最新 ANSI/ISO C++ 草案标准的主要部分,因此每个 C++ 编译器的大厂家都将实现 STL。STL 必将广泛使用。

在 C 和“原始 C++”中,我们用指针访问数组的各个元素。而在 C++ STL 中,则通过迭代器对象访问容器元素,这种形式很像指针,但更加智能化。迭代器类可以在任何容器中通用。

容器封装了一些基本操作,而 STL 算法的实现是与容器无关的。

STL 不用 new 和 delete,而用分配器分配和释放存储空间。程序员可以提供分配器,指定容器如何处理存储管理,但 STL 提供的默认分配器即可以满足大多数应用程序的要求。自定义分配器是个高级课题,不在本书讨论之列。

本章是对 STL 的简介,而不是完整或综合介绍。但这一章足以说明 STL 的价值,鼓励读者进一步钻研。本章也许是本书中最能体现复用价值的一章。STL 容器包括最常用和最重要的数据结构,它们都是模板化的,可以根据特定应用程序相关的数据类型进行调整。

第 15 章介绍了数据结构,建立了链表、队列、堆栈和树,通过指针组成了链接对象。基于指针的代码很复杂,稍有不慎即可能产生错误,造成严重的内存访问破坏和内存泄漏错误。实现队列、优先级队列、集合、映射等其他数据结构需要相当大的工作量。

### 软件工程视点 20.2

不要事事从头开始,而要利用 C++ 标准库的可复用组件编程。STL 中有许多常用数据结构容器,提供了处理这些容器中数据的各种常见算法程序。

### 测试与调试提示 20.1

编写基于指针的数据结构和算法时,需自己测试和调试以保证数据结构、类和算法的工作正确。在这种低层次上操作指针很容易出错,很容易造成严重的内存访问破坏和内存泄漏错误。对大多数程序员及其要编写的大多数应用程序,用 STL 提供的模板化数据结构就足够了。利用 STL 的代码能节省大量测试和调试时间。但是对于大项目,模板编译的时间可能较长。

每个 STL 容器都有相关联的成员函数。有些功能适用于所有 STL 容器,有些是特定容器独有的。我们用类模板 vector、list 和 deque 演示最常见的功能。将在其他 STL 容器例子中介绍容器特定的功能。

我们搜索了大量 Internet/World Wide Web 资源,并将其放在本章末尾,并提供了 STL 相关文献清单。

## 20.1.1 容器简介

图 20.1 显示了 STL 容器类型。容器分为三大类:顺序容器(sequence container)、关联容器(associative container)和容器适配器(container adapter)。顺序容器也称为顺序性容器(sequential container),我们通常称其为顺序容器。顺序容器和关联容器统称为第一类容器(first-class container)。还有另外四种容器称为近容器(near container):C 语言式数组(见第 4 章)、string(见第 19 章)、操作 I/O 标志值的 bitset 和进行高速数学矢量运算的 valarray(这个类经过性能优化,不如第一类容器那么灵活)。这四种容器称为近容器,因为它们与第一类容器提供类似功能,但没有提供第一类容器的全部功能。

标准库容器类	说明
<b>顺序性容器</b>	
vector	从后面快速插入与删除 直接访问任何元素
deque	从前面或后面快速插入与删除 直接访问任何元素
list	双链表, 从任何地方快速插入与删除
<b>关联容器</b>	
set	快速查找, 不允许重复值
multiset	快速查找, 允许重复值
map	一对一映射, 基于关键字快速查找, 不允许重复值
multimap	一对多映射, 基于关键字快速查找, 允许重复值
<b>容器适配器</b>	
stack	后进先出 (LIFO)
queue	先进先出 (FIFO)
priority_queue	最高优先级元素总是第一个出列

图 20.1 标准库容器类

STL 经过认真设计, 使容器提供类似的功能。有许多一般化操作是所有容器都适用的 (如 size 函数), 而有些操作则适用于类似容器的子集。这样就可以用新类扩展 STL。图 20.2 列出的所有标准库容器共有的函数。注意, 重载运算符函数 `operator<`、`operator<=`、`operator>`、`operator>=`、`operator==` 和 `operator!=` 不适用于 `priority_queue`。

所有标准库容器共有的函数	说明
默认构造函数	提供容器默认初始化的构造函数。通常, 每个容器都有几个构造函数, 提供容器初始化的不同方法
复制构造函数	将容器初始化为现有同类容器副本的构造函数
析构函数	不再需要容器时进行内存整理的析构函数
empty	容器中没有元素时返回 true, 否则返回 false
max_size	返回容器中最大元素个数
size	返回容器中当前元素个数
operator=	将一个容器赋给另一个容器
operator<	如果第一个容器小于第二个容器, 则返回 true, 否则返回 false
operator<=	如果第一个容器小于或等于第二个容器, 则返回 true, 否则返回 false
operator>	如果第一个容器大于第二个容器, 则返回 true, 否则返回 false
operator>=	如果第一个容器大于或等于第二个容器, 则返回 true, 否则返回 false
operator==	如果第一个容器等于第二个容器, 则返回 true, 否则返回 false
operator!=	如果第一个容器不等于第二个容器, 则返回 true, 否则返回 false
swap	交换两个容器的元素
<b>只在第一类容器中的函数</b>	
begin	该函数的两个版本返回 iterator 或 const_iterator, 引用容器第一个元素
end	该函数的两个版本返回 iterator 或 const_iterator, 引用容器最后一个元素后面一位
rbegin	该函数的两个版本返回 rverse_iterator 或 const_reverse_iterator, 引用容器最后一个元素
rend	该函数的两个版本返回 rverse_iterator 或 const_reverse_iterator, 引用容器第一个元素前面一位
erase	从容器中清除一个或几个元素
clear	从容器中清除所有元素

图 20.2 所有标准库容器共有的函数

图 20.3 显示了标准库容器的头文件。这些头文件的内容都在 `namespace std` 中。(注意, 有些 C++ 编译器不支持新式的头文件。许多编译器提供自己的头文件名。关于编译器提供的 STL 支持, 参见编译器文档。)

标准库容器头文件	
<code>&lt;vector&gt;</code>	
<code>&lt;list&gt;</code>	
<code>&lt;deque&gt;</code>	
<code>&lt;queue&gt;</code>	包含 <code>queue</code> 和 <code>priority_queue</code>
<code>&lt;stack&gt;</code>	
<code>&lt;map&gt;</code>	包含 <code>map</code> 和 <code>multimap</code>
<code>&lt;set&gt;</code>	包含 <code>set</code> 和 <code>multiset</code>
<code>&lt;bitset&gt;</code>	

图 20.3 标准库容器的头文件

图 20.4 显示了第一类容器中常用的 `typedef` (生成类型的别名或同义词)。这些 `typedef` 用于变量、函数参数和函数返回值的一般性声明。例如, 每个容器中的 `value_type` 总是表示容器中所存放数值类型的 `typedef`。

typedef	说明
<code>value_type</code>	容器中存放的元素类型
<code>reference</code>	容器中存放的元素类型的引用
<code>const_reference</code>	容器中存放的元素类型的常量引用。这种引用只能读取容器中的元素和进行 <code>const</code> 操作
<code>pointer</code>	容器中存放的元素类型的指针
<code>iterator</code>	指向容器中存放的元素类型的迭代器
<code>const_iterator</code>	指向容器中存放的元素类型的常量迭代器, 只能读取容器中的元素
<code>reverse_iterator</code>	指向容器中存放的元素类型的逆向迭代器。这种迭代器在容器中逆向迭代
<code>const_reverse_iterator</code>	指向容器中存放的元素类型的常量逆向迭代器。这种迭代器在容器中逆向迭代, 只能读取容器中的元素
<code>difference_type</code>	引用相同容器的两个迭代器相减结果的类型 ( <code>list</code> 和关联容器的迭代器没有定义 <code>operator-</code> )
<code>size_type</code>	用于计算容器中项目数和检索顺序容器的类型 (不能对 <code>list</code> 检索)

#### 20.4 第一类容器中常用的 typedef

##### 性能提示 20.3

STL 通常避免继承和虚函数, 而使用模板化的常规编程以达到更好的执行性能。

##### 可移植性提示 20.1

STL 必将成为容器编程的优选方法。STL 编程性能提高代码的可移植性。

##### 性能提示 20.4

必须要了解 STL 组件。选择某个问题的最适合容器能提高性能和减少内存需求。

准备使用 STL 容器时, 一定要保证容器中存放的元素类型支持基本功能的集合。元素插入容器中时, 生成该元素的副本。因此, 该元素类型应提供自己的复制构造函数和赋值运算符 (注意: 这

只在元素类型不适合用默认成员复制方法进行复制时才需要)。另外, 关联容器和许多算法要求对元素进行比较。为此, 元素类型应提供相等运算符(==)和小于运算符(<)。

#### 软件工程视点 20.3

容器中存放的元素一般不需要相等和小于运算符, 除非元素要进行比较。但是, 从模板生成代码时, 有些编译器要求所有模板都要定义, 而有些编译器只要求定义程序中实际使用的模板。

### 20.1.2 迭代器简介

迭代器与指针有许多相同之处, 都用于指向第一类容器的元素(还有其他一些用处)。迭代器保存所操作的特定容器需要的状态信息, 从而实现与每种容器类型相适应的迭代器。然而, 有些迭代器操作在所有容器中是一致的。例如, 复引用运算符(\*)总是复引用迭代器, 以便使用其所指的元素。++运算符返回容器中下一个元素的迭代器(就像数组指针递增到指向数组中下一个元素)。

第一类容器提供成员函数begin()和end()。函数begin()返回指向容器第一个元素的迭代器, 函数end()返回指向容器最后一个元素后面一位的迭代器(是个不存在的元素)。如果迭代器i指向特定元素, 则++i指向下一个元素, \*i指向i所指的元素。

我们用iterator类型的对象指可以修改的容器元素, 用const\_iterator类型的对象所指的是不可以修改的容器元素。

我们通过序列(或称为排列)使用迭代器。这些序列可能在容器中, 也可能是输入序列或输出序列。图 20.5 演示了用istream\_iterator输入从标准输入(将数值序列输入程序)和用ostream\_iterator输出向标准输出(将数值序列输出程序)。用户从键盘输入两个整数, 程序显示两个整数的和。

```
1 // Fig. 20.5: fig20_05.cpp
2 // Demonstrating input and output with iterators.
3 #include <iostream>
4 #include <iterator>
5
6 using namespace std;
7
8 int main()
9 {
10     cout << "Enter two integers: ";
11
12     istream_iterator< int > inputInt( cin );
13     int number1, number2;
14
15     number1 = *inputInt;    // read first int from standard input
16     ++inputInt;            // move iterator to next input value
17     number2 = *inputInt;    // read next int from standard input
18
19     cout << "The sum is: ";
20
21     ostream_iterator< int > outputInt( cout );
22
23     *outputInt = number1 + number2;    // output result to cout
24     cout << endl;
```

```

25     return 0;
26 )

```

**输出结果:**

```

Enter two integers: 12 25
The sum is: 37

```

图 20.5 演示输入与输出流迭代器

第 12 行:

```
istream_iterator< int > inputInt( cin );
```

生成 `istream_iterator`，能够以类型安全方式从标准输入对象 `cin` 输入 `int` 值。第 15 行:

```
number1 = *inputInt; // read first int from standard input
```

复引用迭代器 `inputInt`，从 `cin` 读取第一个整数，并将这个整数赋给 `number1`。注意用复引用运算符 `*` 从与 `inputInt` 相关的流读取数值，这与指针的复引用相似。第 16 行:

```
++inputInt; // move iterator to next input value
```

将迭代器 `inputInt` 移到输入流中的下一个值。

```
number2 = *inputInt; // read next int from standard input
```

从 `inputInt` 中输入下一个值并将其赋给 `number2`。

第 21 行:

```
ostream_iterator< int > outputInt( cout );
```

生成 `ostream_iterator`，能向标准输出流 `cout` 输出 `int` 值。第 23 行:

```
*outputInt = number1 + number2; // output result
```

将 `number1` 和 `number2` 的和赋给 `*outputInt`，从而向 `cout` 输出一个整数。注意赋值语句中通过复引用运算符 `*` 将 `*outputInt` 作为左值使用。如果要用 `outputInt` 输出另一个值，则迭代器要用 `++` 递增（可以用前置自增或后置自增）。

**测试与调试提示 20.2**

任何 `const` 迭代器的复引用运算符 `*` 返回容器元素的 `const` 引用，从而不允许使用非 `const` 成员函数。

**常见编程错误 20.1**

如果要复引用容器范围之外的迭代器，则是个运行时的逻辑错误。特别是不能复引用 `end()` 返回的迭代器。

**常见编程错误 20.2**

要对 `const` 容器生成非 `const` 迭代器是个语法错误。

图 20.6 显示了 STL 使用的迭代器类别。每个迭代器类别有一组特定功能。

图 20.7 显示了迭代器类别层次。按这个层次从上到下，每个迭代器类别支持图中上方迭代器类别的所有功能。这样，顶层的迭代器类别功能最弱，底层的迭代器类别功能最强。注意这不是继承层次。

迭代器类别	说明
输入	从容器中读取元素。输入迭代器只能一次一个元素地向前移动（即从容器开头到容器末尾）。输入迭代器只支持一遍算法，同一输入迭代器不能两次遍历一个序列
输出	向容器写入元素。输出迭代器只能一次一个元素地向前移动。输出迭代器只支持一遍算法，同一输出迭代器不能两次遍历一个序列
正向	组合输入迭代器与输出迭代器的功能，并保留在容器中的位置（作为状态信息）
双向	组合正向迭代器功能与逆向移动的功能（即从容器末尾到容器开头）。正向迭代器支持多遍算法
随机访问	组合双向迭代器的功能与直接访问容器中任何元素的功能，即可以向前或向后跳过任意个元素

图 20.6 迭代器类别

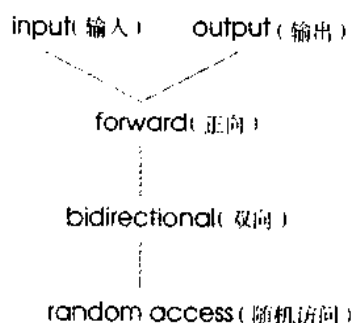


图 20.7 迭代器类别层次

每个容器支持的迭代器类别确定容器能否使用 STL 中的特定算法。支持随机访问迭代器的容器能使用 STL 中的所有算法。可以看出，数组的指针可以在大多数 STL 算法中代替迭代器，包括要求随机访问迭代器的情况。图 20.8 显示了每种 STL 容器支持的迭代器类别。注意只有 vector、deque、list、set、multiset、map 和 multimap（即第一类容器）能用迭代器遍历。

#### 软件工程视点 20.4

使用能达到所需性能的“最弱迭代器”可以产生复用性最强的组件。

容器	支持的迭代器类别
顺序容器	
vector	随机访问
deque	随机访问
list	双向
关联容器	
set	双向
multiset	双向
map	双向
multimap	双向
容器适配器	
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

图 20.8 每种 STL 容器支持的迭代器类别

图 20.9 显示了 STL 容器类定义中的预定义迭代器 typedef。并不是每个容器都定义了一个 typedef。我们用 const 版本的迭代器遍历只读容器，用逆向迭代器逆向遍历容器。

## 测试与调试提示 20.3

对 `const_iterator` 进行操作返回 `const` 引用, 防止修改所操作容器的元素。能用 `const_iterator` 时不要用 `iterator`, 这是最低权限原则的又一个例子。

预定义迭代器 typedef	++ 方向	功能
<code>iterator</code>	向前	读/写
<code>const_iterator</code>	向前	读
<code>reverse_iterator</code>	向后	读/写
<code>const_reverse_iterator</code>	向后	读

图 20.9 预定义迭代器 typedef

图 29.10 显示了每种迭代器类别可以进行的操作。注意每种迭代器类别可以进行的操作包括图中前一种迭代器可以进行的所有操作。另外, 输入和输出迭代器无法保存迭代器, 因此不可以使用保存的值。

迭代器操作	说明
所有迭代器	
<code>++p</code>	前置自增迭代器
<code>p++</code>	后置自增迭代器
输入迭代器	
<code>*p</code>	复引用迭代器, 作为右值
<code>p = p1</code>	将一个迭代器赋给另一个迭代器
<code>p == p1</code>	比较迭代器的相等性
<code>p != p1</code>	比较迭代器的不等性
输出迭代器	
<code>*p</code>	复引用迭代器, 作为左值
<code>p = p1</code>	将一个迭代器赋给另一个迭代器
正向迭代器	正向迭代器提供输入和输出迭代器的所有功能
双向迭代器	
<code>--p</code>	前置自减迭代器
<code>p--</code>	后置自减迭代器
随机访问迭代器	
<code>p += i</code>	将迭代器 <code>p</code> 递增 <code>i</code> 位
<code>p -= i</code>	将迭代器 <code>p</code> 递减 <code>i</code> 位
<code>p + i</code>	在 <code>p</code> 位加 <code>i</code> 位后的迭代器
<code>p - i</code>	在 <code>p</code> 位减 <code>i</code> 位后的迭代器
<code>p[i]</code>	返回 <code>p</code> 位元素偏离 <code>i</code> 位的元素引用
<code>p &lt; p1</code>	如果迭代器 <code>p</code> 小于迭代器 <code>p1</code> (即容器中迭代器 <code>p</code> 在迭代器 <code>p1</code> 之前), 则返回 <code>true</code> , 否则返回 <code>false</code>
<code>p &lt;= p1</code>	如果迭代器 <code>p</code> 小于或等于迭代器 <code>p1</code> (即容器中迭代器 <code>p</code> 在迭代器 <code>p1</code> 之前或 在同一位置), 则返回 <code>true</code> , 否则返回 <code>false</code>
<code>p &gt; p1</code>	如果迭代器 <code>p</code> 大于迭代器 <code>p1</code> (即容器中迭代器 <code>p</code> 在迭代器 <code>p1</code> 之后), 则返回 <code>true</code> , 否则返回 <code>false</code>
<code>p &gt;= p1</code>	如果迭代器 <code>p</code> 大于或等于迭代器 <code>p1</code> (即容器中迭代器 <code>p</code> 在迭代器 <code>p1</code> 之后或 在同一位置), 则返回 <code>true</code> , 否则返回 <code>false</code>

图 20.10 每种迭代器类别可以进行的操作

### 20.1.3 算法简介

STL 的重要方面是提供能在各种容器中通用的算法。STL 提供了许多常用于操作容器的算法，例如插入、删除、查找、排序等等。

STL 提供 70 种左右的标准算法。我们将提供大多数算法的“有生命力的代码”例子并用表格总结其他算法。算法只是间接通过迭代器操作容器元素。许多算法处理迭代器对所定义的一系列元素，第一个迭代器指向序列中第一个元素，第二个迭代器指向序列中最后一个元素。

容器成员函数 `begin()` 返回容器第一个元素的迭代器，`end()` 返回容器最后一个元素后面一位的迭代器。算法通常返回迭代器。

例如，算法 `find()` 寻找一个元素并返回这个元素的迭代器。如果找不到该元素，则 `find()` 返回 `end()` 迭代器（可以用于测试是否找不到该元素）。每个 STL 容都可以使用 `find()` 算法。

#### 软件工程视点 20.5

STL 的实现很简单。到目前为止，类的设计人员要将算法与容器相关联，只需将算法变成容器的成员函数。STL 采用不同的方法。算法与容器是分开的，只是通过迭代器间接操作容器的元素。这样分开可以编写更一般的算法，能适用于许多容器类。

STL 算法提供另一种复用的途径。利用丰富的算法集可以节省程序员的大量时间和精力。

如果算法使用功能更弱的迭代器，则可以用于支持功能更强的迭代器的容器。有些算法要求功能强大的迭代器，例如排序要求随机访问迭代器。

#### 软件工程视点 20.6

STL 是可扩展的，很容易增加新算法而不改变 STL 容器。

#### 软件工程视点 20.7

STL 算法可以处理 STL 容器，也可以处理基于指针的 C 语言式数组。

#### 可移植性提示 20.2

由于 STL 算法只是通过迭代器间接处理容器，一个算法通常可以用于多个不同的容器。

图 20.11 显示了许多变化序列算法（mutating-sequence algorithm），即会修改相应容器的算法。

变化序列算法		
<code>copy()</code>	<code>remove()</code>	<code>reverse_copy()</code>
<code>copy_backward()</code>	<code>remove_copy()</code>	<code>rotate()</code>
<code>fill()</code>	<code>remove_copy_if()</code>	<code>rotate_copy()</code>
<code>fill_n()</code>	<code>remove_if()</code>	<code>stable_partition()</code>
<code>generate()</code>	<code>replace()</code>	<code>swap()</code>
<code>generate_n()</code>	<code>replace_copy()</code>	<code>swap_ranges()</code>
<code>iter_swap()</code>	<code>replace_copy_if()</code>	<code>transform()</code>
<code>partition()</code>	<code>replace_if()</code>	<code>unique()</code>
<code>random_shuffle()</code>	<code>reverse()</code>	<code>unique_copy()</code>

图 20.11 变化序列算法

图 20.12 显示许多非变化序列算法，即不修改相应容器的算法。

图 20.13 显示了头文件 `<numeric>` 的数字型算法。



非变化序列算法		
<code>adjacent-find()</code>	<code>equal()</code>	<code>mismatch()</code>
<code>count()</code>	<code>find()</code>	<code>search()</code>
<code>count_if()</code>	<code>for_each()</code>	<code>search_n()</code>

图 20.12 非变化序列算法

头文件<numeric>的数字型算法
<code>accumulate()</code>
<code>inner_product()</code>
<code>partial_sum()</code>
<code>adjacent_difference()</code>

图 20.13 头文件&lt;numeric&gt;的数字型算法

## 20.2 顺序容器

C++ 标准模板库提供三种顺序容器: `vector`、`list` 和 `deque`。`vector` 类和 `deque` 类都是基于数组的。`list` 类实现链表数据结构, 与第 15 章介绍的 `List` 类相似, 但会使程序更加健壮。

STL 最常用的容器之一是 `vector`。`vector` 类是第 8 章生成的智能 `Array` 类的改进, `vector` 类可以动态改变长度。与 C 和 C++ “原始” 数组 (见第 4 章) 不同的是, `vector` 可以相互赋值。这在基于指针的 C 语言式数组中是不可能的, 因为这些数组名是常量指针, 不能作为赋值目标。和 C 语言数组一样, `vector` 下标不进行自动范围检查, 但 `vector` 类通过成员函数 `at` 提供这个功能。

### 性能提示 20.5

从 `vector` 的后面插入数据十分有效。`vector` 在增加新项目时相应增长。在 `vector` 中间插入或删除的成本很高, 因为插入 (或删除) 元素之后的整个 `vector` 部分都要移动, `vector` 元素和 C 或 C++ “原始” 数组一样占用内存中的连续单元。

图 20.2 显示了所有 STL 容器共同的操作。除了这些操作之外, 每个容器通常还提供各种其他功能。许多功能是几种容器共有的。但这些操作对每个容器的效果并不相同。程序员通常要选择具体应用中最有效的操作。

### 性能提示 20.6

需要在容器两端经常插入 (或删除) 元素的应用程序通常选择 `deque` 而不是选择 `vector`。尽管 `deque` 和 `vector` 都可以在容器两端插入 (或删除) 元素, 但在前面插入 (或删除) 元素时, `deque` 类比 `vector` 类更有效。

### 性能提示 20.7

需要在容器中间或两边经常插入 (或删除) 元素的应用程序通常选择 `list`, 它能有效地在数据结构的任意位置实现插入和删除。

除了图 20.2 介绍的共同操作之外, 顺序容器还有几个共同的操作, `front` 返回容器中第一个元素的引用, `back` 返回容器中最后一个元素的引用, `push_back` 在容器末尾插入新元素, `pop_back` 在容器末尾删除新元素。

### 20.2.1 vector 顺序容器

vector 类提供了具有连续内存地址的数据结构。这样就可以像 C 或 C++ “原始” 数组一样通过下标运算符[]直接、有效地访问矢量的任何元素。类 vector 在容器中的数据需要排序和通过下标方便地访问时最常用。vector 的内存用尽时，vector 自动分配更大的连续内存区，将原先的元素复制到新的内存区，并释放旧的内存区。

#### 性能提示 20.8

vector 容器最适用于达到性能最好的随机访问。

#### 性能提示 20.9

vector 类对象可以用重载下标运算符[]快速索引访问，因为它们具有像 C 或 C++ “原始” 数组一样的连续内存地址。

#### 性能提示 20.10

同时插入多个元素比一次插入一个要快。

每个容器的一个重要部分是所支持的迭代器类型，其确定容器可以采用哪种算法。vector 支持随机访问迭代器，即图 20.10 所示的所有迭代器操作都适用于 vector 迭代器。所有 STL 算法都可以对 vector 进行操作。vector 的迭代器通常实现为 vector 元素的指针。每个取迭代器参数的 STL 算法要求这些迭代器提供最基本的功能。例如，如果算法要求正向迭代器，则这个算法可以作用于任何提供正向迭代器、双向迭代器或随机访问迭代器的容器。只要容器支持算法的基本迭代器功能，该算法就可以作用于这个容器。

图 20.14 演示了 vector 类模板的几个函数。许多函数都是每个标准库的第一类容器所使用的。只有包含<vector>头文件才能使用 vector 类。

```
1 // Fig. 20.14: fig20_14.cpp
2 // Testing Standard Library vector class template
3 #include <iostream>
4 #include <vector>
5
6 using namespace std;
7
8 template < class T >
9 void printVector( const vector< T > &vec );
10
11 int main()
12 {
13     const int SIZE = 6;
14     int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
15     vector< int > v;
16
17     cout << "The initial size of v is: " << v.size()
18         << "\nThe initial capacity of v is: " << v.capacity();
19     v.push_back( 2 ); // method push_back() is in
20     v.push_back( 3 ); // every sequence collection
21     v.push_back( 4 );
22     cout << "\nThe size of v is: " << v.size()
23         << "\nThe capacity of v is: " << v.capacity();
24     cout << "\n\nContents of array a using pointer notation: ";
```

```

25
26     for ( int *ptr = a; ptr != a + SIZE; ++ptr )
27         cout << *ptr << ' ';
28
29     cout << "\nContents of vector v using iterator notation: ";
30     printVector( v );
31
32     cout << "\nReversed contents of vector v: ";
33
34     vector< int >::reverse_iterator p2;
35
36     for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
37         cout << *p2 << ' ';
38     cout << endl;
39     return 0;
40 }
41
42 template < class T >
43 void printVector( const vector< T > &vec )
44 {
45     vector< T >::const_iterator p1;
46
47     for ( p1 = vec.begin(); p1 != vec.end(); p1++ )
48         cout << *p1 << ' ';
49 }

```

**输出结果:**

```

The initial size of v is: 0
The initial capacity of v is: 0
The size of v is: 3
The capacity of v is: 4

```

```

Contents of array a using pointer notation: 1 2 3 4 5 6
Contents of vector v using iterator notation: 2 3 4
Reversed contents of vector v: 4 3 2

```

图 20.14 演示标准库 vector 类模板

**第15行:**

```
vector< int > v;
```

声明 vector 类的实例 v，存放 int 值。实例化这个对象时，生成一个长度为 0 的空 vector（即 vector 中存放的元素个数为 0），容量为 0（不增加内存时 vector 可以存放的元素个数）。

**第17行和第18行:**

```

cout << "The initial size of v is: " << v.size()
    << "\nThe initial capacity of v is: " << v.capacity();

```

演示本例中的 vector v 的 size 和 capacity 函数最初都返回 0。每个容器中都有函数 size，该函数返回容器中当前存放的元素个数。函数 capacity 返回不增加内存时容器可以存放的元素个数。

**第19行到第21行:**

```
v.push_back( 2 ); // method push_back() is in
```

```
v.push_back( 3 ); // every sequence container
v.push_back( 4 );
```

用函数 `push_back` (所有顺序容器中都有) 在 `vector` 末尾增加元素 (即下一个可用位置)。如果在已满的 `vector` 中增加元素, 则 `vector` 自动增加长度, 有些 STL 版本自动将 `vector` 的长度翻倍。

#### 性能提示 20.11

需要更多空间时自动将 `vector` 的长度翻倍可能会浪费内存。例如, 一个 1 000 000 元素的已满 `vector` 要增加一个元素时, 如果自动将 `vector` 的长度翻倍, 则可以放 2 000 000 个元素, 其中有 999 999 个元素是未用的。程序员可以用 `resize()` 更好地控制内存空间的使用。

第 22 行和第 23 行用 `size` 和 `capacity` 演示 `push_back` 操作之后 `vector` 的新长度与新容量。函数 `size` 返回 3, 即加进 `vector` 的元素个数。函数 `capacity` 返回 4, 表示 `vector` 不必增加内存即可再增加一个元素。增加第一个元素时, `v` 的长度变为 1, 容量变为 1。增加第二个元素时, `v` 的长度变为 2, 容量变为 2。增加第三个元素时, `v` 的长度变为 3, 容量变为 4。如果再增加两个元素, 则 `v` 的长度变为 5, 容量变为 8。每次 `vector` 已满而增加另一个元素时, 总容量增加一倍。

第 26 行和 27 行演示如何用指针和指针算法输出数组内容。第 30 行调用 `printVector` 函数, 用迭代器输出 `vector` 内容。函数模板 `printVector` 的定义从第 43 行开始。函数接收对 `vector` 的 `const` 引用作为参数。第 45 行:

```
vector< T >::const_iterator p1;
```

声明一个称为 `p1` 的 `const_iterator`, 对 `vector` 进行迭代并输出其内容。`const_iterator` 使程序可以读取 `vector` 的元素, 但不让程序修改其元素。第 47 行和第 48 行的 `for` 结构:

```
for ( p1 = vec.begin(); p1 != vec.end(); p1++ )
    cout << *p1 << ' ';
```

用 `vector` 成员函数 `begin` 初始化 `p1`, 返回 `vector` 中第一个元素的 `const_iterator` (另一种 `begin` 版本返回 `iterator`, 可以用于非 `const` 容器)。只要 `p1` 不超过 `vector` 末尾, 继续循环。这是通过比较 `p1` 与 `vec.end()` 的结果确定的, 其返回一个 `const_iterator` (和 `begin` 一样, 还有另一个 `end` 版本返回 `iterator`), 表示 `vector` 最后一个元素的后一位。如果 `p1` 等于这个值, 则已经到达 `vector` 末尾。所有第一类容器都有函数 `begin` 和 `end`。循环体复引用迭代器 `p1`, 取得 `vector` 中当前元素的值。表达式 `p1++` 将迭代器放在 `vector` 的下一个元素上。

#### 测试与调试提示 20.4

只有随机访问迭代器支持 `<`, 最好用 `!=` 和 `end()` 测试容器末尾。

第 34 行:

```
vector< int >::reverse_iterator p2;
```

声明一个 `reverse_iterator`, 可以逆向对 `vector` 进行迭代。所有第一类容器都支持这种迭代器。

第 36 行和 37 行:

```
for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
    cout << *p2 << ' ';
```

用类似于函数 `printVector` 的 `for` 结构对 `vector` 迭代。在这个循环中，函数 `rbegin`（即容器中逆向迭代的开始迭代器）和 `rend`（即容器中逆向迭代的结束迭代器）指定逆向输出的元素范围。和 `begin` 和 `end` 函数一样，`rbegin` 和 `rend` 可以根据容器是否为常量而返回 `const_reverse_iterator` 或 `reverse_iterator`。

图 20.15 演示的函数可以读取和操作 `vector` 的元素。第 14 行：

```
vector< int > v( a, a + SIZE );
```

用重载的 `vector` 构造函数，该函数取两个迭代器参数。记住，数组指针可以作为迭代器。这个语句生成整型 `vector` `v`，并用地址 `a` 到地址 `a + SIZE`（不包括 `a + SIZE`）的整型数组 `a` 将其初始化。

```
1 // Fig. 20.15: fig20_15.cpp
2 // Testing Standard Library vector class template
3 // element-manipulation functions
4 #include <iostream>
5 #include <vector>
6 #include <algorithm>
7
8 using namespace std;
9
10 int main()
11 {
12     const int SIZE = 6;
13     int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
14     vector< int > v( a, a + SIZE );
15     ostream_iterator< int > output( cout, " " );
16     cout << "Vector v contains: ";
17     copy( v.begin(), v.end(), output );
18
19     cout << "\nFirst element of v: " << v.front()
20          << "\nLast element of v: " << v.back();
21
22     v[ 0 ] = 7;          // set first element to 7
23     v.at( 2 ) = 10;      // set element at position 2 to 10
24     v.insert( v.begin() + 1, 22 ); // insert 22 as 2nd element
25     cout << "\nContents of vector v after changes: ";
26     copy( v.begin(), v.end(), output );
27
28     try {
29         v.at( 100 ) = 777; // access element out of range
30     }
31     catch ( out_of_range e ) {
32         cout << "\nException: " << e.what();
33     }
34
35     v.erase( v.begin() );
36     cout << "\nContents of vector v after erase: ";
37     copy( v.begin(), v.end(), output );
38     v.erase( v.begin(), v.end() );
39     cout << "\nAfter erase, vector v "
40          << ( v.empty() ? "is" : "is not" ) << " empty";
41
42     v.insert( v.begin(), a, a + SIZE );
43     cout << "\nContents of vector v before clear: ";
44     copy( v.begin(), v.end(), output );
```

```

45     v.clear(); // clear calls erase to empty a container
46     cout << "\nAfter clear, vector v "
47         << ( v.empty() ? "is" : "is not" ) << " empty";
48
49     cout << endl;
50     return 0;
51 }

```

#### 输出结果:

```

Vector v contains: 1 2 3 4 5 6
First element of v: 1
Last element of v: 6
Contents of vector v after changes: 7 22 2 10 4 5 6
Exception: invalid vector<T> subscript
Contents of vector v after erase: 22 2 10 4 5 6
After erase, vector v is empty
Contents of vector v after erase: 1 2 3 4 5 6
After clear, vector v is empty

```

图 20.15 演示标准库 vector 类模板的元素操作函数

#### 第 15 行:

```
ostream_iterator< int > output( cout, " " );
```

声明一个称为 output 的 ostream\_iterator, 可以通过 cout 输出用一个空格分隔的整数。ostream\_iterator 是个类型安全的输出机制, 只输出 int 类型或兼容类型的值。构造函数的第一个参数指定输出流, 第二个参数是个字符串, 指定输出值的分隔字符, 这里是空格符。我们用 ostream\_iterator 输出 vector 内容。

#### 第 17 行:

```
copy( v.begin(), v.end(), output );
```

用标准库中的 copy 算法将 vector v 的整个内容输出到标准输出。算法 copy 将容器中指定位置 ( 由第一个参数中的迭代器指定 ) 的每个元素复制到第二个参数中迭代器指定的位置 ( 并不包括该位置 )。第一个和第二个参数必须符合输入迭代器要求, 即通过这个迭代器能从容器中读取数值。元素复制到作为最后一个参数的输出迭代器 ( 即通过这个迭代器能存放或输出数值 ) 指定的地址。这里, 输出迭代器是连接 cout 的 ostream\_iterator output, 因此元素复制到标准输出。只有包含头文件 <algorithm> 才能使用标准库的算法。

第 19 行和第 20 行用 front 和 back 函数 ( 所有顺序容器都有 ) 分别确定 vector 的第一个和最后一个元素。

#### 常见编程错误 20.3

vector 不能是空的, 否则 front 和 back 函数的结果为未定义。

#### 第 22 行和第 23 行:

```

v[ 0 ] = 7;           // set first element to 7
v.at( 2 ) = 10;       // set element at position 2 to 10

```

演示 `vector` 下标的两种写法 (也适用于 `deque` 容器)。第 22 行用下标运算符, 重载成返回指定位置数值的引用或常量引用 (根据容器是否为常量)。函数 `at` 进行相同的操作, 但增加了边界检查特性。函数 `at` 首先检查参数提供的值, 确定是否在 `vector` 的边界之内。如果不是, 则 `at` 函数抛出 `out_of_bounds` 异常 (见第 28 行到第 33 行)。图 20.16 显示了一些 STL 异常类型 (第 13 章“异常处理”介绍了标准库异常类型)。

STL 异常类型	说明
<code>out_of_range</code>	表示下标超界, 例如 <code>vector</code> 成员函数 <code>at</code> 指定无效下标
<code>invalid_argument</code>	表示向函数传入了无效参数
<code>length_error</code>	表示试图生成太长的容器、 <code>string</code> 等
<code>bad_alloc</code>	表示用 <code>new</code> (或分配器) 分配内存失败, 因为内存不足

20.16 STL 异常类型

第 24 行:

```
v.insert( v.begin() + 1, 22 ); // insert 22 as 2nd element
```

使用每个顺序容器都有的三个 `insert` 函数之一。上述语句在第一个参数中的迭代器所指定位置的元素之前插入数值 22。本例中, 迭代器指向 `vector` 的第二个元素, 因此 22 插入为第二个元素, 原先的第二个元素变为 `vector` 的第三个元素。其他版本的 `insert` 可以从容器中特定位置开始插入同一个值的多个副本或从容器中特定位置开始插入另一个容器 (或数组) 的一组值。

第 35 行和第 38 行:

```
v.erase( v.begin() );
v.erase( v.begin(), v.end() );
```

使用两个 `erase` 函数 (所有第一类容器中都有)。第 35 行表示应从容器中清除迭代器参数所指定位置的元素 (本例中为 `vector` 开头的元素)。第 38 行指定应从容器中清除从第一个参数指定的位置开始到第二个参数指定的位置 (但不包括该位置) 的所有元素, 本例中, 应从 `vector` 清除所有元素。第 40 行用函数 `empty` (包含适配器的所有容器都有) 确认 `vector` 为空容器。

#### 常见编程错误 20.4

清除包含指向动态分配对象的指针的元素并不删除这个对象。

第 42 行:

```
v.insert( v.begin(), a, a + SIZE );
```

使用 `insert` 函数, 该函数第二和第三个参数指定要插入 `vector` 的序列值的开始位置与结束位置 (也许来自另一个容器, 但这里来自整型数组 `a`)。记住, 结束位置指定要插入的最后一个元素后一个位置, 程序一直复制到这个位置, 但不包括这个位置。

最后, 第 45 行:

```
v.clear(); // clear calls erase to empty a container
```

用函数 `clear` (所有第一类容器都有) 清空 `vector`。这个函数调用第 38 行所用的 `erase` 版本实际进行这个操作。

注意: 还有其他所有容器和所有顺序容器共有的函数, 这里不可能一一介绍。我们将在下面几节介绍其中大多数函数, 还将介绍许多每个容器特有的函数。

### 20.2.2 list 顺序容器

list 顺序容器提供在容器中任何位置进行插入与删除操作的有效实现方法。如果大多数插入和删除发生在容器末尾, 则 20.2.3 节介绍的 deque 数据结构能提供更有效实现方法。list 类是个双链表, 即 list 中的每个节点包含 list 中上一节点和下一个节点的指针。这样, list 类支持双向迭代器, 使容器可以正向和逆向遍历。任何需要输入、输出、正向和双向迭代器的算法都可以作用于 list。list 的许多成员函数将容器元素作为顺序集合进行操作。

除了图 20.2 中所有 STL 容器的成员函数和图 20.5 中所有顺序容器的成员函数之外, list 类还提供八个成员函数 splice、push\_front、pop\_front、remove、unique、merge、reverse 和 sort。图 20.17 演示了 list 类的几个特性。记住, 图 20.14 和图 20.15 所列的许多函数都适用于 list 类。只有包含头文件 <list> 才能使用 list 类。

```
1 // Fig. 20.17: fig20_17.cpp
2 // Testing Standard Library class list
3 #include <iostream>
4 #include <list>
5 #include <algorithm>
6
7 using namespace std;
8
9 template < class T >
10 void printList( const list< T > &listRef );
11
12 int main()
13 {
14     const int SIZE = 4;
15     int a[ SIZE ] = { 2, 6, 4, 8 };
16     list< int > values, otherValues;
17
18     values.push_front( 1 );
19     values.push_front( 2 );
20     values.push_back( 4 );
21     values.push_back( 3 );
22
23     cout << "values contains: ";
24     printList( values );
25     values.sort();
26     cout << "\nvalues after sorting contains: ";
27     printList( values );
28
29     otherValues.insert( otherValues.begin(), a, a + SIZE );
30     cout << "\notherValues contains: ";
31     printList( otherValues );
32     values.splice( values.end(), otherValues );
33     cout << "\nAfter splice values contains: ";
34     printList( values );
35
36     values.sort();
```



```

37  cout << "\nvalues contains: ";
38  printList( values );
39  otherValues.insert( otherValues.begin(), a, a + SIZE );
40  otherValues.sort();
41  cout << "\notherValues contains: ";
42  printList( otherValues );
43  values.merge( otherValues );
44  cout << "\nAfter merge:\n  values contains: ";
45  printList( values );
46  cout << "\n  otherValues contains: ";
47  printList( otherValues );
48
49  values.pop_front();
50  values.pop_back(); // all sequence containers
51  cout << "\nAfter pop_front and pop_back values contains:\n";
52  printList( values );
53
54  values.unique();
55  cout << "\nAfter unique values contains: ";
56  printList( values );
57
58  // method swap is available in all containers
59  values.swap( otherValues );
60  cout << "\nAfter swap:\n  values contains: ";
61  printList( values );
62  cout << "\n  otherValues contains: ";
63  printList( otherValues );
64
65  values.assign( otherValues.begin(), otherValues.end() );
66  cout << "\nAfter assign values contains: ";
67  printList( values );
68
69  values.merge( otherValues );
70  cout << "\nvalues contains: ";
71  printList( values );
72  values.remove( 4 );
73  cout << "\nAfter remove( 4 ) values contains: ";
74  printList( values );
75  cout << endl;
76  return 0;
77 }
78
79 template < class T >
80 void printList( const list< T > &listRef )
81 {
82     if ( listRef.empty() )
83         cout << "List is empty";
84     else {
85         ostream_iterator< T > output( cout, " " );
86         copy( listRef.begin(), listRef.end(), output );
87     }
88 }

```

**输出结果:**

values contains: 2 1 4 3

```

values after sorting contains: 1 2 3 4
otherValues contains: 2 4 6 8
After splice values contains: 1 2 3 4 2 6 4 8
values contains: 1 2 2 3 4 4 6 8
otherValues contains: 2 4 6 8
After merge:
    values contains: 1 2 2 2 3 4 4 4 6 6 8 8
    otherValues contains: List is empty
After pop_front and pop_back values contains:
2 2 2 3 4 4 4 6 6 8
After unique values contains: 2 3 4 6 8
After swap:
    values contains: List is empty
    otherValues contains: 2 3 4 6 8
After assign values contains: 2 3 4 6 8
values contains: 2 2 3 3 4 4 6 6 8 8
After remove( 4 ) values contains: 2 2 3 3 6 6 8 8

```

图 20.17 演示标准库 list 类模板

第 16 行:

```
list< int > values, otherValues;
```

实例化两个 list 对象，可以存放整数。第 18 行和第 19 行用函数 `push_front` 在 `values` 开头插入整数。函数 `push_front` 是类 list 与 deque（不包括 vector）特有的。第 20 行和第 21 行用函数 `push_back` 在 `values` 末尾插入整数。记住，函数 `push_back` 是所有顺序容器共有的函数。

第 25 行:

```
values.sort();
```

用 list 成员函数 `sort` 按升序排列 list 中的元素。（注意：这与 STL 算法中的 `sort` 不同。）`sort` 的另一种形式允许程序员提供二元判定函数，该判定函数取两个参数（链表中的两个值）进行比较，并返回一个 bool 值，表示比较结果。`sort` 函数确定 list 元素排列的顺序，这个版本对存放指针而不是存放数值的 list 特别有用。（说明：我们在图 20.28 中演示一元判定函数。一元判定函数取一个参数，用该参数进行比较，并返回一个 bool 值，表示比较结果。）

第 32 行:

```
values.splice( values.end(), otherValues );
```

用 list 函数 `splice` 删除 `otherValues` 中的元素，并将其插入 `values` 中第一个参数指定的迭代器位置之前。这个函数还有另外两个版本。有三个参数的 `splice` 函数可以从第二个参数指定的容器中删除第三个参数的迭代器指定位置的元素。有四个参数的 `splice` 函数可以从第二个参数指定的容器中删除第三和第四个参数的迭代器指定范围内的元素，并放在第一个参数指定的位置。

在 list `otherValues` 中插入元素并排序 `values` 和 `otherValues` 之后，第 43 行:

```
values.merge( otherValues );
```

用 list 成员函数 `merge` 删除 `otherValues` 的所有元素并将其按顺序插入 `values` 中。两个 list 要先按相同顺序排序之后才能进行这个操作。`merge` 的第二个版本允许程序员提供一个判定函数，该判定函数取两个参数（list 中的值）并返回一个 bool 值。判定函数指定 `merge` 使用的排列顺序。

第 49 行用 `list` 函数 `pop_front` 删除 `list` 中的第 1 个元素。第 50 行用函数 `pop_back` (所有顺序容器都有) 删除 `list` 中的最后 1 个元素。

第 54 行:

```
values.unique();
```

用 `list` 函数 `unique` 删除 `list` 中的重复元素。`list` 要先进行排序之后才能进行这个操作, 使重复项目放在一起, 便于删除重复元素。`unique` 的第二个版本允许程序员提供一个判定函数, 该函数取两个参数 (`list` 中的值) 并返回一个 `bool` 值。判定函数确定两个值是否相等。

第 59 行:

```
values.swap( otherValues );
```

用函数 `swap` (所有容器都有) 交换 `values` 与 `otherValues` 的内容。

第 65 行:

```
values.assign( otherValues.begin(), otherValues.end() );
```

用 `list` 函数 `assign` 将 `values` 内容换成 `otherValues` 的内容, 替换在两个迭代器参数指定的范围内。`assign` 的第二个版本将原内容换成第二个参数指定值的副本。函数的第一个参数指定副本数。

第 72 行:

```
values.remove( 4 );
```

用 `list` 函数 `remove` 删除 `list` 中数值 4 的所有副本。

### 20.2.3 deque 顺序容器

`deque` 类在一个容器中提供了 `vector` 和 `list` 的许多好处。`deque` 是 "double-ended queue" (双头队列) 的缩写。`deque` 类能利用下标提供有效的索引访问, 可以像 `vector` 一样读取与修改元素。`deque` 类还像 `list` 一样能有效地在前面和后面进行插入与删除操作 (但 `list` 还能在中间进行插入与删除操作)。`deque` 类还支持随机访问迭代器, 因此 `deque` 可以使用所有 STL 算法。`deque` 的最常见用法之一是维护先进先出的元素队列。

需要增加 `deque` 的存储空间时, 可以在内存块中 `deque` 两端进行分配, 通常保存为这些块的指针数组。由于 `deque` 利用非连续内存布局, 因此 `deque` 迭代器要比 `vector` 和基于指针数组中用于迭代的指针更加智能化。

#### 性能提示 20.12

对 `deque` 分配存储块之后, 有些版本中要等删除 `deque` 时才释放这个块。这样就使 `deque` 比重复分配、释放和再分配内存块时更有效。但与此同时, `deque` 也更浪费内存 (例如与 `vector` 相比)。

#### 性能提示 20.13

在 `deque` 中间进行插入与删除操作已经过优化, 通过减少要复制的元素个数可以保持 `deque` 元素的连续性。

`deque` 类提供了 `vector` 类的一些基本操作, 还增加成员函数 `push_front` 和 `pop_front`, 分别在 `deque` 开头插入和删除。

图 20.18 演示了 `deque` 类的特性。记住, 图 20.14、20.15 和 20.17 中的许多函数也适用于 `deque` 类。只有包含头文件 `<deque>` 才能使用 `deque` 类。

```
1 // Fig. 20.18: fig20_18.cpp
2 // Testing Standard Library class deque
3 #include <iostream>
4 #include <deque>
5 #include <algorithm>
6
7 using namespace std;
8
9 int main()
10 {
11     deque< double > values;
12     ostream_iterator< double > output( cout, " " );
13
14     values.push_front( 2.2 );
15     values.push_front( 3.5 );
16     values.push_back( 1.1 );
17
18     cout << "values contains: ";
19
20     for ( int i = 0; i < values.size(); ++i )
21         cout << values[ i ] << ' ';
22
23     values.pop_front();
24     cout << "\nAfter pop_front values contains: ";
25     copy ( values.begin(), values.end(), output );
26
27     values[ 1 ] = 5.4;
28     cout << "\nAfter values[ 1 ] = 5.4 values contains: ";
29     copy ( values.begin(), values.end(), output );
30     cout << endl;
31     return 0;
32 }
```

**输出结果:**

```
values contains: 3.5 2.2 1.1
After pop_front values contains: 2.2 1.1
After values[ 1 ] = 5.4 values contains: 2.2 5.4
```

图 20.18 演示标准库 deque 类模板

**第 11 行:**

```
deque< double > values;
```

实例化一个 deque，可以存放 double 值。第 14 行到第 16 行用函数 push\_front 和 push\_back 在 deque 的开头和结尾插入元素。记住，push\_back 适用于所有顺序容器，而 push\_front 只适用于 list 和 deque 类。

**第 20 行的 for 结构:**

```
for ( int i = 0; i < values.size(); ++i )
    cout << values[ i ] << ' ';
```

用下标运算符读取 deque 中每个元素的值以便输出。注意本例中用函数 size 确保不访问 deque 边界之外的元素。

第 23 行用函数 `pop_front` 演示从 `deque` 中删除第一个元素。记住 `push_front` 只适用于 `list` 和 `deque` 类，而不适用于 `vector` 类。

第 27 行：

```
values[ 1 ] = 5.4;
```

用下标运算符生成一个左值，这样就可以直接向 `deque` 的任何元素赋值。

## 20.3 关联容器

STL 关联容器能通过关键字（也称查找关键字，`search key`）直接访问从而存储和读取元素。这四个关联容器是 `multiset`、`set`、`multimap` 和 `map`。在关联容器中，按排序顺序维护关键字。对关联容器迭代时，按该容器的排列顺序遍历。`multiset` 和 `set` 类提供了控制数值集合的操作，其中数值是关键字，即不必另有一个值与每个关键字相关联。`multiset` 和 `set` 的主要差别在于 `multiset` 允许重复键而 `set` 不允许重复键。`multimap` 和 `map` 类提供了操作与每个关键字相关联的值的方法（这些值也称为映射值，`mapped value`）。`multiset` 和 `map` 的主要差别在于 `multimap` 允许存放与数值相关联的重复关键字，而 `map` 只允许存放与数值相关联的惟一关键字。除了图 20.2 显示的所有容器的共同成员函数之外，所有关联容器还支持另外几个成员函数，包括 `find`、`lower_bound`、`upper_bound` 和 `count`。下面几节介绍每个关联容器和关联容器共同的成员函数的例子。

### 20.3.1 `multiset` 关联容器

`multiset` 关联容器用于快速存储和读取关键字。`multiset` 允许重复关键字。元素的顺序由比较器函数对象（`comparator function object`）确定。例如，在整型 `multiset` 中，元素可以按升序排列，只要用比较器函数对象 `less<int>` 排序关键字。在所有关联容器中，该关键字的数据类型必须正确地支持比较，这种比较是由比较器函数对象指定的，使用 `less<int>` 排序的关键字必须支持 `operator<` 的比较。如果关联容器为程序员定义的数据类型，则这些类型应提供相应的比较运算符。`multiset` 支持双向迭代器（但不支持随机访问迭代器）。

#### 性能提示 20.14

由于性能原因，`multiset` 和 `set` 通常实现为所谓的红黑折半查找树。利用这种内部表达方法，折半查找树通常是平衡的，从而减少平均查找时间。

图 20.19 演示了按升序排列的整型 `multiset` 中的 `multiset` 关联容器。只有包含头文件 `<set>` 才能使用 `multiset`。`multiset` 和 `set` 容器提供相同的成员函数。

```
1 // Fig. 20.19: fig20_19.cpp
2 // Testing Standard Library class multiset
3 #include <iostream>
4 #include <set>
5 #include <algorithm>
6
7 using namespace std;
8
9 int main()
10 {
```

```

11  const int SIZE = 10;
12  int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
13  typedef multiset< int, less< int > > ims;
14  ims intMultiset;    // ims for "integer multiset"
15  ostream_iterator< int > output( cout, " " );
16
17  cout << "There are currently " << intMultiset.count( 15 )
18       << " values of 15 in the multiset\n";
19  intMultiset.insert( 15 );
20  intMultiset.insert( 15 );
21  cout << "After inserts, there are "
22       << intMultiset.count( 15 )
23       << " values of 15 in the multiset\n";
24
25  ims::const_iterator result;
26
27  result = intMultiset.find( 15 ); // find returns iterator
28
29  if ( result != intMultiset.end() ) // if iterator not at end
30      cout << "Found value 15\n";    // found search value 15
31
32  result = intMultiset.find( 20 );
33
34  if ( result == intMultiset.end() ) // will be true hence
35      cout << "Did not find value 20\n"; // did not find 20
36
37  intMultiset.insert( a, a + SIZE ); // add array a to multiset
38  cout << "After insert intMultiset contains:\n";
39  copy( intMultiset.begin(), intMultiset.end(), output );
40
41  cout << "\nLower bound of 22: "
42       << *( intMultiset.lower_bound( 22 ) );
43  cout << "\nUpper bound of 22: "
44       << *( intMultiset.upper_bound( 22 ) );
45
46  pair< ims::const_iterator, ims::const_iterator > p;
47
48  p = intMultiset.equal_range( 22 );
49  cout << "\nUsing equal_range of 22"
50       << "\n  Lower bound: " << *( p.first )
51       << "\n  Upper bound: " << *( p.second );
52  cout << endl;
53  return 0;
54 )

```

**输出结果:**

```

There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset
Found value 15
Did not find value 20
After insert intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100
lower bound of 22: 22
upper bound of 22: 30
Using equal_range of 22

```

```
Lower bound: 22
Upper bound: 30
```

图 20.19 演示标准库 `multiset` 类模板

第 13 行和第 14 行:

```
typedef multiset< int, less< int > > ims;
ims intMultiset;    // ims for "integer multiset"
```

用 `typedef` 生成用函数对象 `less< int >` 按升序排列的整型 `multiset` 类型。这个新类型用于实例化一个整型 `multiset` 对象 `intMultiset`。

#### 编程技巧 20.1

用 `typedef` 使得类型名较长的代码 (如 `multiset`) 更易读。

第 17 行的输出语句:

```
cout << "There are currently " << intMultiset.count( 15 )
      << " values of 15 in the multiset\n";
```

用函数 `count` (所有关联容器都有) 计算当前 `multiset` 中数值 15 出现的次数。

第 19 行到第 20 行:

```
intMultiset.count( 15 );
intMultiset.count( 15 );
```

用 `insert` 函数的三个版本之一将数值 15 加进 `multiset` 中两次。`insert` 的第二个版本取迭代器和数值参数, 在指定的迭代器位置开始查找插入点。第三个 `insert` 版本取两个迭代器参数, 确定一个数值范围, 将另一个容器中的元素加进 `multiset` 中。

第 27 行:

```
result = intMultiset.find( 15 ); // find returns iterator
```

用函数 `find` (所有关联容器都有) 找到 `multiset` 中的数值 15。函数 `find` 返回 `iterator` 或 `const_iterator`, 指向第一个找到数值的位置。如果找不到数值, 则 `find` 返回的 `iterator` 或 `const_iterator` 等于调用 `end` 返回的值。

第 37 行:

```
intMultiset.insert( a, a + SIZE ); // add array a to multiset
```

用函数 `insert` 将数组 `a` 的元素插入 `multiset` 中。第 39 行中, `copy` 算法将 `multiset` 的元素复制到标准输出中。注意元素按升序显示。

第 41 行到第 44 行:

```
cout << "\nLower bound of 22: "
      << *( intMultiset.lower_bound( 22 ) );
cout << "\nUpper bound of 22: "
      << *( intMultiset.upper_bound( 22 ) );
```

用函数 `lower_bound` 和 `upper_bound` (所有关联容器都有) 确定 `multiset` 中第一个找到数值 22 的位置和最后一个找到数值 22 位置的后一个位置。两个函数都返回 `iterator` 或 `const_iterator`, 指向相应地址, 如果找不到数值, 则返回值等于调用 `end` 返回的值。

第 46 行:

```
pair< ims::const_iterator, ims::const_iterator > p;
```

实例化 `pair` 类的对象 `p`。 `pair` 类对象用于操作数值对。本例中, `pair` 的内容是整型 `multiset` 的两个 `const_iterator`。 `p` 的目的是存放 `multiset` 函数 `equal_range` 的返回值, 其返回的 `pair` 包含 `lower_bound` 和 `upper_bound` 操作的结果。 `pair` 类型包含两个 `public` 数据成员 `first` 和 `second`。

第 48 行:

```
p = intMultiset.equal_range( 22 );
```

用函数 `equal_range` 确定 `multiset` 中数值 22 的 `lower_bound` 和 `upper_bound`。第 50 行和第 51 行分别用 `p.first` 和 `p.second` 访问 `lower_bound` 和 `upper_bound`。我们通过复引用迭代器将 `equal_range` 返回的地址中的数值输出。

### 20.3.2 set 关联容器

`set` 关联容器用于快速存储和读取惟一的关键词。`set` 的实现方法与 `multiset` 相似, 只是 `set` 要用惟一的关键词。因此, 如果在 `set` 中插入重复关键词, 则忽略重复值, 由于这是 `set` 的数学行为, 因此不作为常见编程错误。`set` 支持双向迭代器 (但不支持随机访问迭代器)。图 20.20 演示了双精度类型的 `set`。只有包含 `<set>` 头文件才能使用 `set` 类。

```
1 // Fig. 20.20: fig20_20.cpp
2 // Testing Standard Library class set
3 #include <iostream>
4 #include <set>
5 #include <algorithm>
6
7 using namespace std;
8
9 int main()
10 {
11     typedef set< double, less< double > > double_set;
12     const int SIZE = 5;
13     double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
14     double_set doubleSet( a, a + SIZE );
15     ostream_iterator< double > output( cout, " " );
16
17     cout << "doubleSet contains: ";
18     copy( doubleSet.begin(), doubleSet.end(), output );
19
20     pair< double_set::const_iterator, bool > p;
21
22     p = doubleSet.insert( 13.8 ); // value not in set
23     cout << '\n' << *( p.first )
24         << ( p.second ? " was" : " was not" ) << " inserted";
25     cout << "\ndoubleSet contains: ";
26     copy( doubleSet.begin(), doubleSet.end(), output );
```



```

27
28     p = doubleSet.insert( 9.5 ); // value already in set
29     cout << '\n' << *( p.first )
30         << ( p.second ? " was" : " was not" ) << " inserted";
31     cout << "\ndoubleSet contains: ";
32     copy( doubleSet.begin(), doubleSet.end(), output );
33
34     cout << endl;
35     return 0;
36 }

```

**输出结果:**

```

doubleSet contains: 2.1 3.7 4.2 9.5
13.8 was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8
9.5 was not inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

```

图 20.20 演示标准库 set 类模板

第 11 行:

```
typedef set< double, less< double > > double_set;
```

用 typedef 生成 double 值集合的新类型, 用函数对象 less< double > 按升序排列。第 14 行:

```
double_set doubleSet( a, a + SIZE );
```

用新类型 double\_set 实例化对象 doubleSet。构造函数调用取数组 a 中到 a + SIZE 的元素 (即整个数组) 并将其插入 set 中。第 18 行用 copy 算法输出 set 内容。注意数值 2.1 (在数组 a 中出现两次) 只在 doubleSet 中出现一次, 这是因为容器 set 不允许重复值。

第 20 行:

```
pair< double_set::const_iterator, bool > p;
```

声明一个 pair, 包括 double\_set 的 const\_iterator 和一个 bool 值。这个对象存放调用 set 函数 insert 的结果。

第 22 行:

```
p = doubleSet.insert( 13.8 ); // value not in set
```

用函数 insert 将数值 13.8 放在 set 中。返回的 pair、p 包含一个迭代器 p.first, 指向 set 中的数值 13.8; 还有一个 bool 值, 在数值插入时为 true, 在数值没有插入时为 false (已经在 set 中)。

### 20.3.3 multimap 关联容器

multimap 关联容器用于快速存储和读取关键字与相关值 (通常称为关键字/数值对, key/value pair)。multiset 和 set 使用的许多方法也适用于 multimap 和 map。multimap 和 map 的元素是关键字和数值对而不是各个值。插入 multimap 和 map 时, 使用包含关键字和数值对的 pair 对象。关键字的顺序由比较器函数对象确定。例如, 在用整数作为关键字类型的 multimap 中, 关键字可以按升序排列, 只要用比较器函数对象 less< int > 排序关键字。multimap 中允许重复关键字, 因此一个关键字可以对应多个值, 称为一对多关系。例如, 在信用卡事务处理系统中, 一个信用卡账号可能有许多

相关联的事务；在大学中，一个学生可能选多门课，一个教授可能教多个学生；在军队中，一个衔级的人员有许多。multimap 支持双向迭代器（但不支持随机访问迭代器）。与 multiset 和 set 一样，multimap 通常实现为红黑二叉查找树，树的节点是关键字/数值对。图 20.21 演示了 multimap 关联容器。只有包含头文件 <map> 才能使用 multimap 类。

```
1 // Fig. 20.21: fig20_21.cpp
2 // Testing Standard Library class multimap
3 #include <iostream>
4 #include <map>
5
6 using namespace std;
7
8 int main()
9 {
10     typedef multimap< int, double, less< int > > mmid;
11     mmid pairs;
12
13     cout << "There are currently " << pairs.count( 15 )
14         << " pairs with key 15 in the multimap\n";
15     pairs.insert( mmid::value_type( 15, 2.7 ) );
16     pairs.insert( mmid::value_type( 15, 99.3 ) );
17     cout << "After inserts, there are "
18         << pairs.count( 15 )
19         << " pairs with key 15\n";
20     pairs.insert( mmid::value_type( 30, 111.11 ) );
21     pairs.insert( mmid::value_type( 10, 22.22 ) );
22     pairs.insert( mmid::value_type( 25, 33.333 ) );
23     pairs.insert( mmid::value_type( 20, 9.345 ) );
24     pairs.insert( mmid::value_type( 5, 77.54 ) );
25     cout << "Multimap pairs contains:\nKey\tValue\n";
26
27     for ( mmid::const_iterator iter = pairs.begin();
28         iter != pairs.end(); ++iter )
29         cout << iter->first << '\t'
30             << iter->second << '\n';
31
32     cout << endl;
33     return 0;
34 }
```

**输出结果：**

```
There are currently 0 pairs with kdy 15 in the multimap
After inserts, there are 2 pairs with key 15
Multimap pairs contains:
Key      Value
5        77.54
10       22.22
15       2.7
15       99.3
20       9.345
25       33.333
30       111.11
```

图 20.21 演示标准库 multimap 类模板

**性能提示 20.15**

`multimap` 的实现可以有效地对指定关键字匹配所有值。

第 10 行:

```
typedef multimap< int, double, less< int > > mmid;
```

用 `typedef` 定义 `multimap` 类型, 其中关键字类型为 `int`, 相关值类型为 `double`, 元素按升序排列。第 11 行用新类型实例化一个称为 `pairs` 的 `multimap`。

第 13 行的语句:

```
cout << "There are currently " << pairs.count( 15 )
      << " pairs with key 15\n";
```

用函数 `count` 确定关键字为 15 的关键字/数值对个数。

第 15 行:

```
pairs.insert( mmid::value_type( 15, 2.7 ) );
```

用函数 `insert` 在 `multimap` 中增加新的关键字/数值对。表达式 `mmid::value_type(15, 2.7)` 生成 `pair` 对象, 其中 `first` 是 `int` 类型的关键字 (15), `second` 是 `double` 类型的值 (2.7)。类型 `mmid::value_type` 在第 10 行定义为 `multimap` 的 `typedef` 的一部分。

第 27 行的 `for` 结构输出 `multimap` 内容, 包括关键字和数值。第 29 行和第 30 行:

```
cout << iter->first << '\t'
      << iter->second << '\n';
```

用一个称为 `iter` 的 `const_iterator` 访问 `multimap` 每个元素中的 `pair` 成员。注意输出中的关键字是按升序排列的。

### 20.3.4 map 关联容器

`map` 关联容器用于快速存储和读取关键字与相关值。`map` 中不允许重复关键字, 因此一个关键字只可以对应一个值, 称为一对一映射。例如, 公司用 100、200、300 之类的惟一员工号时, 可以用 `map` 将员工号与内线电话 4321、4115、5217 相关联。利用 `map` 可以指定关键字, 迅速取得相关数据。`map` 通常称为关联数组 (associative array)。在 `map` 的下标运算符 `[]` 中提供关键字, 即可找到 `map` 中该关键字的相关数据。可以在 `map` 中任何地方进行插入与删除。

图 20.22 演示了 `map` 关联容器。图 20.22 使用了与图 20.21 相似的特性, 但使用下标运算符。只有包含头文件 `<map>` 才能使用 `map` 类。第 29 行和第 30 行:

```
pairs[ 25 ] = 9999.99; // change existing value for 25
pairs[ 40 ] = 8765.43; // insert new value for 40
```

使用 `map` 类的下标运算符。当下标是 `map` 中的现有关键字时, 运算符返回相关值的引用。当下标是 `map` 中没有的关键字时, 运算符在 `map` 中插入关键字并返回一个引用, 可以将一个值与该关键字相关联。第 29 行将关键字 25 的值 (首先在第 7 行指定为 33.333) 换成新值 9999.99。第 30 行在 `map` 中插入新的关键字/数值对 (称为建立关联)。

```

1 // Fig. 20.22: fig20_22.cpp
2 // Testing Standard Library class map
3 #include <iostream>
4 #include <map>
5
6 using namespace std;
7
8 int main()
9 {
10     typedef map< int, double, less< int > > mid;
11     mid pairs;
12
13     pairs.insert( mid::value_type( 15, 2.7 ) );
14     pairs.insert( mid::value_type( 30, 111.11 ) );
15     pairs.insert( mid::value_type( 5, 1010.1 ) );
16     pairs.insert( mid::value_type( 10, 22.22 ) );
17     pairs.insert( mid::value_type( 25, 33.333 ) );
18     pairs.insert( mid::value_type( 5, 77.54 ) ); // dupe ignored
19     pairs.insert( mid::value_type( 20, 9.345 ) );
20     pairs.insert( mid::value_type( 15, 99.3 ) ); // dupe ignored
21     cout << "pairs contains:\nKey\tValue\n";
22
23     mid::const_iterator iter;
24
25     for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
26         cout << iter->first << '\t'
27             << iter->second << '\n';
28
29     pairs[ 25 ] = 9999.99; // change existing value for 25
30     pairs[ 40 ] = 8765.43; // insert new value for 40
31     cout << "\nAfter subscript operations, pairs contains:"
32         << "\nKey\tValue\n";
33
34     for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
35         cout << iter->first << '\t'
36             << iter->second << '\n';
37
38     cout << endl;
39     return 0;
40 }

```

**输出结果:**

```

pairs contains:
Key      Value
5        1010.1
10       22.22
15       2.7
20       9.345
25       33.333
30       111.11

```

After subscript operations, pairs contains:

```

Key      Value
5        1010.1
10       22.22

```

15	2.7
20	9.345
25	9999.99
30	111.11
40	8765.43

图 20.22 演示标准库 map 类模板

## 20.4 容器适配器

STL 提供了三个容器适配器 (container adapter) —— stack、queue 和 priority\_queue。容器适配器不是第一类容器, 因为它们不提供存放数据的实际数据结构的实现方法, 而且容器适配器不支持迭代器。容器适配器的好处在于程序员可以选择相应的基础数据结构。所有的三个容器适配器都提供成员函数 push 和 pop, 以实现在容器适配器数据结构中插入元素的方法和从容器适配器数据结构中读取元素的方法。下面几节介绍适配器类的例子。

### 20.4.1 stack 适配器

stack 类提供的功能可以从基础数据结构的末尾插入或删除 (通常称为后进先出的数据结构)。stack 可以用任何顺序容器 vector、list 和 deque 实现。本例用标准库中的每个顺序容器生成一个整数堆栈, 表示 stack。默认情况下, stack 用 deque 实现。stack 的操作有: push 在 stack 顶上插入一个元素 (调用基础容器的 push\_back 函数实现), pop 从 stack 顶上删除一个元素 (调用基础容器的 pop\_back 函数实现), top 取得 stack 顶上元素的引用 (调用基础容器的 back 函数实现), empty 确定 stack 是否为空 (调用基础容器的 empty 函数实现), size 取得 stack 的元素个数 (调用基础容器的 size 函数实现)。

#### 性能提示 20.16

stack 的每个常见操作都实现为内联函数, 调用基础容器的相应函数。这样可以避免二次函数调用的开销。

#### 性能提示 20.17

为了达到最佳性能, 用 deque 或 vector 类作为 stack 的基础容器。

图 20.23 演示 stack 适配器类。只有包含头文件 <stack> 才能使用 stack 类。

```
1 // Fig. 20.23: fig20_23.cpp
2 // Testing Standard Library class stack
3 #include <iostream>
4 #include <stack>
5 #include <vector>
6 #include <list>
7
8 using namespace std;
9
10 template< class T >
11 void popElements( T &s );
12
13 int main()
```

```

14 {
15     stack< int > intDequeStack; // default is deque-based stack
16     stack< int, vector< int > > intVectorStack;
17     stack< int, list< int > > intListStack;
18
19     for ( int i = 0; i < 10; ++i ) {
20         intDequeStack.push( i );
21         intVectorStack.push( i );
22         intListStack.push( i );
23     }
24
25     cout << "Popping from intDequeStack: ";
26     popElements( intDequeStack );
27     cout << "\nPopping from intVectorStack: ";
28     popElements( intVectorStack );
29     cout << "\nPopping from intListStack: ";
30     popElements( intListStack );
31
32     cout << endl;
33     return 0;
34 }
35
36 template< class T >
37 void popElements( T &s )
38 {
39     while ( !s.empty() ) {
40         cout << s.top() << ' ';
41         s.pop();
42     }
43 }

```

**输出结果:**

```

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0

```

图 20.13 演示标准库 stack 适配器类模板

第 15 行到第 17 行:

```

stack< int > intDequeStack; // default is deque-based stack
stack< int, vector< int > > intVectorStack;
stack< int, list< int > > intListStack;

```

实例化三个整数堆栈。第 15 行指定的整数 stack 用默认 deque 容器作为基础数据结构。第 16 行指定的整数 stack 用整数 vector 作为基础数据结构。第 17 行指定的整数 stack 用整数 list 作为基础数据结构。

第 20 行到第 22 行各用一个 push 函数（每个适配器类都有）将整数放在每个 stack 顶上。

第 37 行的函数 popElements 将每个 stack 中的元素弹出。第 40 行用 stack 函数 top 读取 stack 顶上的元素以便输出。函数 top 不删除顶上的元素。第 41 行用函数 pop（每个适配器类都有）删除顶上的元素。函数 pop 不返回数值。

### 20.4.2 queue 适配器

queue 类提供的功能可以从基础数据结构的结尾插入或在开头删除 (通常称为先进先出数据结构)。queue 可以用 STL 数据结构 list 和 deque 实现。默认情况下, queue 用 deque 实现。queue 的操作有: push 在 queue 后面插入一个元素 (调用基础容器的 push\_back 函数实现), pop 从 queue 前面删除一个元素 (调用基础容器的 pop\_front 函数实现), front 取得 queue 第一个元素的引用 (调用基础容器的 front 函数实现), back 取得 queue 最后一个元素的引用 (调用基础容器的 back 函数实现), empty 确定 queue 是否为空 (调用基础容器的 empty 函数实现), size 取得 queue 的元素个数 (调用基础容器的 size 函数实现)。

#### 性能提示 20.18

queue 的每个常见操作都实现为内联函数, 调用基础容器的相应函数。这样可以避免二次函数调用的开销。

#### 性能提示 20.19

为了达到最佳性能, 用 deque 类作为 queue 的基础容器。

图 20.24 演示 queue 适配器类。只有包含头文件 <queue> 才能使用 queue 类。

```
1 // Fig. 20.24: fig20_24.cpp
2 // Testing Standard Library adapter class template queue
3 #include <iostream>
4 #include <queue>
5
6 using namespace std;
7
8 int main()
9 {
10     queue< double > values;
11
12     values.push( 3.2 );
13     values.push( 9.8 );
14     values.push( 5.4 );
15
16     cout << "Popping from values: ";
17
18     while ( !values.empty() ) {
19         cout << values.front() << ' '; // does not remove
20         values.pop();                  // removes element
21     }
22
23     cout << endl;
24     return 0;
25 }
```

#### 输出结果:

Popping from values: 3.2 9.8 5.4

图 20.24 演示标准库 queue 适配器类模板

第 10 行:

```
queue< double > values;
```

实例化存放 double 值的 queue。第 12 行到第 14 行用函数 push 将元素增加到 queue 中。第 18 行的 while 结构用函数 empty（所有容器都有）确定 queue 是否为空。

当 queue 中有更多元素时，第 19 行用 queue 函数 front 读取（而不删除）queue 中第一个元素以便输出。第 20 行用函数 pop（所有适配器类都有）删除 queue 中第一个元素。

### 20.4.3 priority\_queue 适配器

priority\_queue 提供的功能可以按排列顺序插入基础数据结构并从基础数据结构的前面删除。priority\_queue 可以用 STL 数据结构 vector 和 deque 实现。默认情况下，priority\_queue 实现时用 vector 基础数据结构。在 priority\_queue 中增加元素时，该元素自动按优先顺序插入，使最高优先级元素（即最大值）首先从 priority\_queue 中取出。这通常是用堆排序（heapsort）技术实现的，该技术是把最大值（即最高优先级）元素放在数据结构前面，这种数据结构称为堆（heap）。默认的元素比较是用比较器函数对象 less< T> 进行的，但程序员可以提供其他比较器。

常见 priority\_queue 操作有：push 根据 priority\_queue 的优先顺序将元素插入相应位置（先调用基础容器的 push\_back 函数，然后用堆排序重新排列元素），pop 从 priority\_queue 中删除最大值（即最高优先级）元素（删除堆顶上的元素之后调用基础容器的 pop\_back 函数），top 取得 priority\_queue 中顶上元素的引用（调用基础容器的 front 函数），empty 确定 priority\_queue 是否为空（调用基础容器的 empty 函数），size 取得 priority\_queue 的元素个数（调用基础容器的 size 函数）。

#### 性能提示 20.20

priority\_queue 的每个常见操作都实现为内联函数，调用基础容器的相应函数。这样可以避免二次函数调用的开销。

#### 性能提示 20.21

为了达到最佳性能，用 vector 类作为 priority\_queue 的基础容器。

图 20.25 演示 priority\_queue 适配器类。只有包含头文件 <queue> 才能使用 priority\_queue 类。

```
1 // Fig. 20.25: fig20_25.cpp
2 // Testing Standard Library class priority_queue
3 #include <iostream>
4 #include <queue>
5 #include <functional>
6
7 using namespace std;
8
9 int main()
10 {
11     priority_queue< double > priorities;
12
13     priorities.push( 3.2 );
14     priorities.push( 9.8 );
15     priorities.push( 5.4 );
16
17     cout << "Popping from priorities: ";
18
19     while ( !priorities.empty() ) {
```



```

20     cout << priorities.top() << ' ';
21     priorities.pop();
22 }
23
24     cout << endl;
25     return 0;
26 }

```

**输出结果:**

Popping from priorities: 9.8 5.4 3.2

图 20.25 演示标准库 priority\_queue 适配器类

第11行:

```
priority_queue< double > priorities;
```

实例化存放 double 值的 priority\_queue 并用 deque 作为基础数据结构。第13行到第15行用函数 push 将元素增加到 priority\_queue 中。第19行的 while 结构用函数 empty( 所有容器都有 ) 确定 priority\_queue 是否为空。当 priority\_queue 中有更多元素时, 第20行用 priority\_queue 函数 top 读取 priority\_queue 中最高优先级元素以便输出。第21行用函数 pop( 所有适配器类都有 ) 删除 priority\_queue 中最高优先级元素。

## 20.5 算法

在 STL 之前, 不同厂家的容器和算法类库实际上是不兼容的。早期容器库通常使用继承和多态, 会有虚函数调用的相关开销。早期的库将算法作为类行为放在类库中。STL 将算法与容器分开, 使得更容易加入新算法。这样实现 STL 能更加有效, 避免了虚函数调用的相关开销。利用 STL, 容器元素可以通过迭代器访问。

### 软件工程视点 20.8

STL 算法不依赖于所操作容器的实现细节。只要容器 ( 或数组 ) 的迭代器符合算法要求, STL 算法即可像处理 STL 容器一样处理任何 C 语言式、基于指针的数组 ( 以及用户自定义的数据结构 )。

### 软件工程视点 20.9

不必修改容器类就可以方便地将算法加进 STL 中。

### 20.5.1 fill、fill\_n、generate 与 generate\_n

图 20.26 演示标准库函数 fill、fill\_n、generate 与 generate\_n。函数 fill 和 fill\_n 将容器中一定范围的元素设置为特定值。函数 generate 和 generate\_n 用产生器函数 ( generator function ) 生成容器中一定范围的元素值。产生器函数不取参数, 并返回可以放在容器元素中的值。

```

1 // Fig. 20.26: fig20_26.cpp
2 // Demonstrating fill, fill_n, generate, and generate_n
3 // Standard Library methods.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>

```

```

7
8 using namespace std;
9
10 char nextLetter();
11
12 int main()
13 {
14     vector< char > chars( 10 );
15     ostream_iterator< char > output( cout, " " );
16
17     fill( chars.begin(), chars.end(), '5' );
18     cout << "Vector chars after filling with 5s:\n";
19     copy( chars.begin(), chars.end(), output );
20
21     fill_n( chars.begin(), 5, 'A' );
22     cout << "\nVector chars after filling five elements"
23           << " with As:\n";
24     copy( chars.begin(), chars.end(), output );
25
26     generate( chars.begin(), chars.end(), nextLetter );
27     cout << "\nVector chars after generating letters A-J:\n";
28     copy( chars.begin(), chars.end(), output );
29
30     generate_n( chars.begin(), 5, nextLetter );
31     cout << "\nVector chars after generating K-O for the"
32           << " first five elements:\n";
33     copy( chars.begin(), chars.end(), output );
34
35     cout << endl;
36     return 0;
37 }
38
39 char nextLetter()
40 {
41     static char letter = 'A';
42     return letter++;
43 }

```

**输出结果:**

```

vector chars after filling with 5s:
5 5 5 5 5 5 5 5 5 5
vector chars after filling five elements with As:
A A A A A 5 5 5 5 5
vector chars after generating letters A-J:
A B C D E F G H I J
vector chars after generating K-O for the first five elements:
K L M N O F G H I J

```

图 20.16 演示标准库函数 fill、fill\_n、generate 与 generate\_n

**第 17 行:**

```
fill( chars.begin(), chars.end(), '5' );
```

用函数 `fill` 将字符 'S' 放在 `vector chars` 从 `chars.begin()` 到 `chars.end()` (不包括 `chars.end()`) 的每个元素中。注意, 第一个和第二个参数提供的迭代器至少应为正向迭代器 (可以正向从容器输入或向容器输出)。

第 21 行:

```
fill_n( chars.begin(), 5, 'A' );
```

用函数 `fill_n` 将字符 'A' 放在 `vector chars` 的前五个元素中。第一个参数提供的迭代器至少应为输出迭代器 (可以正向向容器输出)。第二个参数指定填充的元素个数。第三个参数指定每个元素采用的值。

第 26 行:

```
generate( chars.begin(), chars.end(), nextLetter );
```

用函数 `generate` 将调用产生器函数 `nextLetter` 得到的结果放在 `vector chars` 中从 `chars.begin()` 到 `chars.end()` (但不包括 `chars.end()`) 的每个元素中。第一、第二个参数提供的迭代器至少应为正向迭代器。函数 `nextLetter` (在第 39 行定义) 根据 `static` 局部变量中维护的字符 'A' 开始执行。第 42 行:

```
return letter++;
```

返回每次调用 `nextLetter` 时的当前 `letter` 值, 然后递增 `letter` 值。

第 30 行:

```
generate_n( chars.begin(), 5, nextLetter );
```

用函数 `generate_n` 将调用产生器函数 `nextLetter` 得到的结果放在 `vector char` 中从 `chars.begin()` 开始的 5 个元素中。第一个参数提供的迭代器至少应为输出迭代器。

## 20.5.2 equal、mismatch 和 lexicographical\_compare

图 20.27 演示标准库函数 `equal`、`mismatch` 和 `lexicographical_compare`, 比较序列值的相等性。

```
1 // Fig. 20.27: fig20_27.cpp
2 // Demonstrates standard library functions equal,
3 // mismatch, lexicographical_compare.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 int main()
11 {
12     const int SIZE = 10;
13     int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14     int a2[ SIZE ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
15     vector< int > v1( a1, a1 + SIZE ),
16                   v2( a1, a1 + SIZE ),
17                   v3( a2, a2 + SIZE );
18     ostream_iterator< int > output( cout, " " );
19
20     cout << "Vector v1 contains: ";
```

```

21  copy( v1.begin(), v1.end(), output );
22  cout << "\nVector v2 contains: ";
23  copy( v2.begin(), v2.end(), output );
24  cout << "\nVector v3 contains: ";
25  copy( v3.begin(), v3.end(), output );
26
27  bool result = equal( v1.begin(), v1.end(), v2.begin() );
28  cout << "\n\nVector v1 " << ( result ? "is" : "is not" )
29        << " equal to vector v2.\n";
30
31  result = equal( v1.begin(), v1.end(), v3.begin() );
32  cout << "Vector v1 " << ( result ? "is" : "is not" )
33        << " equal to vector v3.\n";
34
35  pair< vector< int >::iterator,
36        vector< int >::iterator > location;
37  location = mismatch( v1.begin(), v1.end(), v3.begin() );
38  cout << "\nThere is a mismatch between v1 and v3 at "
39        << "location " << ( location.first - v1.begin() )
40        << "\nwhere v1 contains " << *location.first
41        << " and v3 contains " << *location.second
42        << "\n\n";
43
44  char c1[ SIZE ] = "HELLO", c2[ SIZE ] = "BYE BYE";
45
46  result =
47    lexicographical_compare( c1, c1 + SIZE, c2, c2 + SIZE );
48  cout << c1
49        << ( result ? " is less than " : " is greater than " )
50        << c2;
51
52  cout << endl;
53  return 0;
54 )

```

#### 输出结果:

```

Vector v1 contains: 1 2 3 4 5 6 7 8 9 10
Vector v2 contains: 1 2 3 4 5 6 7 8 9 10
Vector v3 contains: 1 2 3 4 1000 6 7 8 9 10

```

```

Vector v1 is equal to vector v2.
Vector v1 is not equal to vector v3.

```

```

There is a mismatch between v1 and v3 at location 4
Where v1 contains 5 and v2 contains 1000

```

```

HELLO is greater than BYE BYE

```

图 20.27 演示标准库函数 `equal`、`mismatch` 和 `lexicographical_compare`

#### 第 27 行:

```

bool result = equal( v1.begin(), v1.end(), v2.begin() );

```

用函数 `equal` 比较两个数值序列的相等性。每个序列不一定包含相同的元素个数，如果两个序列的元素个数不同，则 `equal` 返回 `false`。函数 `operator==` 进行元素的比较。本例中，vector `v1` 中从 `v1.begin()` 到 `v1.end()` (不包括 `v1.end()`) 的元素与 vector `v2` 中从 `v2.begin()` 开始的元素进行比较 (本例中 `v1` 和 `v2` 相等)。三个迭代器参数至少应为输入迭代器 (即可以用于从序列正向输入)。第 31 行用函数 `equal` 比较 vector `v1` 与 `v3`，两者是不相等的。

`equal` 函数还有另一种形式，即取一个二元判定函数作为第 4 个参数。二元判定函数接收两个要比较的元素并返回一个 `bool` 值，表示元素是否相同。这可以在存放数值指针而不是存放实际数值的序列中有用，因为可以定义指针所指项目的比较，而不是比较指针内容 (即指针中存放的地址)。

第 35 行到第 37 行：

```
pair< vector< int >::iterator,
      vector< int >::iterator > location;
location = mismatch( v1.begin(), v1.end(), v3.begin() );
```

首先实例化一个 `pair` 迭代器对，即表示整型 vector 的 `location`。这个对象存放第 37 行调用 `mismatch` 的结果。函数 `mismatch` 比较两个数值序列，返回一个 `pair` 迭代器对，表示每个序列中不匹配元素的地址。如果所有元素匹配，则 `pair` 中的两个迭代器等于每个序列的最后迭代器。三个迭代器参数至少应为输入迭代器。本例中要确定 vector 中不匹配的实际地址，使用第 39 行的表达式 `location.first-v1.begin()`。这个计算的结果是迭代器之间的元素个数 (类似于第 5 章介绍的指针算法)。该结果对应于本例中的元素个数，因为比较是从每个 vector 开头进行的。

和函数 `equal` 一样，`mismatch` 还有另一种形式，即取一个二元判定函数作为第 4 个参数。

第 46 行和第 47 行：

```
result =
    lexicographical_compare( c1, c1 + SIZE, c2, c2 + SIZE );
```

用函数 `lexicographical_compare` 比较两个字符数组的内容。该函数的 4 个迭代器参数至少应为输入迭代器。众所周知，数组指针是随机访问迭代器。前两个迭代器参数指定第一个序列的地址范围，后两个迭代器参数指定第二个序列的地址范围。在序列中迭代时，如果第一个序列中的元素小于第二个序列中的元素，则函数返回 `true`。如果第一个序列中的元素大于或等于第二个序列中的元素，则函数返回 `false`。这个函数可以按词法排序序列，这种序列通常包含字符串。

### 20.5.3 `remove`、`remove_if`、`remove_copy` 和 `remove_copy_if`

图 20.28 演示用标准库函数 `remove`、`remove_if`、`remove_copy` 和 `remove_copy_if` 从序列中删除数值。

```
1 // Fig. 20.28: fig20_28.cpp
2 // Demonstrates Standard Library functions remove, remove_if
3 // remove_copy and remove_copy_if
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 bool greater9( int );
```

```

11
12 int main()
13 {
14     const int SIZE = 10;
15     int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
16     ostream_iterator< int > output( cout, " " );
17
18     // Remove 10 from v
19     vector< int > v( a, a + SIZE );
20     vector< int >::iterator newLastElement;
21     cout << "Vector v before removing all 10s:\n";
22     copy( v.begin(), v.end(), output );
23     newLastElement = remove( v.begin(), v.end(), 10 );
24     cout << "\nVector v after removing all 10s:\n";
25     copy( v.begin(), newLastElement, output );
26
27     // Copy from v2 to c, removing 10s
28     vector< int > v2( a, a + SIZE );
29     vector< int > c( SIZE, 0 );
30     cout << "\n\nVector v2 before removing all 10s "
31           << "and copying:\n";
32     copy( v2.begin(), v2.end(), output );
33     remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
34     cout << "\nVector c after removing all 10s from v2:\n";
35     copy( c.begin(), c.end(), output );
36
37     // Remove elements greater than 9 from v3
38     vector< int > v3( a, a + SIZE );
39     cout << "\n\nVector v3 before removing all elements"
40           << "\ngreater than 9:\n";
41     copy( v3.begin(), v3.end(), output );
42     newLastElement = remove_if( v3.begin(), v3.end(),
43                                greater9 );
44     cout << "\nVector v3 after removing all elements"
45           << "\ngreater than 9:\n";
46     copy( v3.begin(), newLastElement, output );
47
48     // Copy elements from v4 to c,
49     // removing elements greater than 9
50     vector< int > v4( a, a + SIZE );
51     vector< int > c2( SIZE, 0 );
52     cout << "\n\nVector v4 before removing all elements"
53           << "\ngreater than 9 and copying:\n";
54     copy( v4.begin(), v4.end(), output );
55     remove_copy_if( v4.begin(), v4.end(),
56                    c2.begin(), greater9 );
57     cout << "\nVector c2 after removing all elements"
58           << "\ngreater than 9 from v4:\n";
59     copy( c2.begin(), c2.end(), output );
60
61     cout << endl;
62     return 0;
63 }
64

```

```

65 bool greater9( int x )
66 {
67     return x > 9;
68 }

```

**输出结果:**

Vector v before removing all 10s:

10 2 10 4 16 6 14 8 12 10

Vector v after removing all 10s:

2 4 16 6 14 8 12

Vector v2 before removing all 10s and copying:

10 2 10 4 16 6 14 8 12 10

Vector c after removing all 10s from v2:

2 4 16 6 14 8 12 0 0 0

Vector v3 before removing all elements  
greater than 9:

10 2 10 4 16 6 14 8 12 10

Vector v3 after removing all elements  
greater than 9:

2 4 6 8

Vector v4 before removing all elements  
greater than 9 and copying:

10 2 10 4 16 6 14 8 12 10

vector c2 after removing all elements  
greater than 9 from v4:

2 4 6 8 0 0 0 0 0 0

图 20.28 演示标准库函数 `remove`、`remove_if`、`remove_copy` 和 `remove_copy_if`

**第 23 行:**

```
newLastElement = remove( v.begin(), v.end(), 10 );
```

用函数 `remove` 删除 vector `v` 中从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 的范围内所有数值为 10 的元素。前两个迭代器参数应为正向迭代器, 使这个算法能修改序列中的元素。这个函数不修改 vector 中的元素个数, 也不破坏删除的元素, 而是将所有未删除元素放在 vector 开头。函数返回 vector 中最后一个未删除元素后面一位的迭代器。从这个迭代器位置到 vector 末尾的元素数值为未定义 (本例中, 每个未定义位置取值为 0)。

**第 33 行:**

```
remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
```

用函数 `remove_copy` 复制 vector `v2` 中从 `v2.begin()` 到 `v2.end()` (但不包括 `v2.end()`) 的范围内所有数值不为 10 的元素, 这些元素放在 vector `c` 中从 `c.begin()` 位置开始处。前两个迭代器参数应为输入迭代器, 第三个参数提供的迭代器应为输出迭代器, 使复制的元素能够插入复制位置。这个函数返回复制到 vector `c` 的最后一个元素后面一位迭代器。注意第 29 行用矢量构造函数, 接收 vector 中的元素个数和这些元素的初始值。

**第 42 行和第 43 行:**

```
newLastElement = remove_if( v3.begin(), v3.end(),
                             greater9 );
```

用函数 `remove_if` 删除 vector `v3` 中从 `v3.begin()` 到 `v3.end()` (但不包括 `v3.end()`) 的范围内, 用户自定义的一元判定函数 `greater9` 返回 `true` 的所有元素。第 64 行将函数 `greater9` 定义为在传入的数值大于 9 时返回 `true`, 否则返回 `false`。前两个迭代器应为正向迭代器, 使这个算法能修改序列中的元素。这个函数不修改 vector 中的元素个数。但该函数把所有未删除元素移到 vector 开头。这个函数返回未删除的 vector 中最后一个元素后面一位的迭代器。从这个迭代器位置到 vector 末尾的元素数值为未定义。

第 55 行和 56 行:

```
remove_copy_if( v4.begin(), v4.end(),
                c2.begin(), greater9 );
```

用函数 `remove_copy_if` 复制 vector `v4` 中从 `v4.begin()` 到 `v4.end()` (但不包括 `v4.end()`) 的范围内, 用户自定义的一元判定函数 `greater9` 返回 `true` 的所有元素。元素放在 vector `c2` 中从 `c2.begin()` 开始处。前两个迭代器参数应为输入迭代器, 第三个参数提供的迭代器应为输出迭代器, 使复制的元素能够插入复制位置。这个函数返回复制到 vector `c2` 的最后一个元素后面一位迭代器。

#### 20.5.4 `replace`、`replace_if`、`replace_copy` 和 `replace_copy_if`

图 20.29 演示用标准库函数 `replace`、`replace_if`、`replace_copy` 和 `replace_copy_if` 从一个序列替换数值。

```
1 // Fig. 20.29: fig20_29.cpp
2 // Demonstrates Standard Library functions replace, replace_if
3 // replace_copy and replace_copy_if
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 bool greater9( int );
11
12 int main()
13 {
14     const int SIZE = 10;
15     int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
16     ostream_iterator< int > output( cout, " " );
17
18     // Replace 10s in v1 with 100
19     vector< int > v1( a, a + SIZE );
20     cout << "Vector v1 before replacing all 10s:\n";
21     copy( v1.begin(), v1.end(), output );
22     replace( v1.begin(), v1.end(), 10, 100 );
23     cout << "\nVector v1 after replacing all 10s with 100s:\n";
24     copy( v1.begin(), v1.end(), output );
25
26     // copy from v2 to c1, replacing 10s with 100s
27     vector< int > v2( a, a + SIZE );
```



```

28  vector< int > c1( SIZE );
29  cout << "\n\nVector v2 before replacing all 10s "
30      << "and copying:\n";
31  copy( v2.begin(), v2.end(), output );
32  replace_copy( v2.begin(), v2.end(),
33              c1.begin(), 10, 100 );
34  cout << "\nVector c1 after replacing all 10s in v2:\n";
35  copy( c1.begin(), c1.end(), output );
36
37  // Replace values greater than 9 in v3 with 100
38  vector< int > v3( a, a + SIZE );
39  cout << "\n\nVector v3 before replacing values greater"
40      << " than 9:\n";
41  copy( v3.begin(), v3.end(), output );
42  replace_if( v3.begin(), v3.end(), greater9, 100 );
43  cout << "\nVector v3 after replacing all values greater"
44      << " than 9 with 100s:\n";
45  copy( v3.begin(), v3.end(), output );
46
47  // Copy v4 to c2, replacing elements greater than 9 with 100
48  vector< int > v4( a, a + SIZE );
49  vector< int > c2( SIZE );
50  cout << "\n\nVector v4 before replacing all values greater"
51      << " than 9 and copying:\n";
52  copy( v4.begin(), v4.end(), output );
53  replace_copy_if( v4.begin(), v4.end(), c2.begin(),
54                  greater9, 100 );
55  cout << "\nVector c2 after replacing all values greater"
56      << " than 9 in v4:\n";
57  copy( c2.begin(), c2.end(), output );
58
59  cout << endl;
60  return 0;
61 }
62
63 bool geater9( int x )
64 {
65     return x > 9;
66 }

```

**输出结果:**

```

Vector v1 before replacing all 10s:
10 2 10 4 16 6 14 8 12 10
Vector v1 after replacing all 10s with 100s:
100 2 100 4 16 6 14 8 12 100

Vector v2 before replacing all 10s and copying:
10 2 10 4 16 6 14 8 12 10
Vector c1 after replacing all 10s in v2:
100 2 100 4 16 6 14 8 12 100

Vector v3 before replacing values greater than 9:
10 2 10 4 16 6 14 8 12 10
Vector v3 after replacing all values greater
than 9 with 100s:

```

```

100 2 100 4 100 6 100 8 100 100

Vector v4 before replacing all values greater
than 9 and copying:
10 2 10 4 16 6 14 8 12 10
Vector c2 after replacing all values greater
than 9 in v4:
100 2 100 4 100 6 100 8 100 100

```

图 20.29 演示标准库函数 `replace`、`replace_if`、`replace_copy` 和 `replace_copy_if`

第 22 行:

```
replace( v1.begin(), v1.end(), 10, 100 );
```

用函数 `replace` 将新值 100 替换 vector `v1` 中从 `v1.begin()` 到 `v1.end()` (但不包括 `v1.end()`) 的范围内所有数值为 10 的元素。前两个迭代器参数应为正向迭代器, 使这个算法能修改序列中的元素。

第 32 行和第 33 行:

```
replace_copy( v2.begin(), v2.end(),
              c1.begin(), 10, 100 );
```

用函数 `replace_copy` 复制 vector `v2` 中从 `v2.begin()` 到 `v2.end()` (但不包括 `v2.end()`) 的范围内所有元素, 将数值为 10 的元素换成新值 100。元素复制到 vector `c1` 中从 `c1.begin()` 开始的位置。前两个迭代器参数应为输入迭代器, 第三个参数提供的迭代器应为输出迭代器, 使复制的元素能够插入复制位置。这个函数返回复制到 vector `c2` 的最后一个元素后面一位迭代器。

第 42 行:

```
replace_if( v3.begin(), v3.end(), greater9, 100 );
```

用函数 `replace_if` 替换 vector `v3` 中从 `v3.begin()` 到 `v3.end()` (但不包括 `v3.end()`) 的范围内, 用户自定义的一元判定函数 `greater9` 返回 `true` 的所有元素。第 63 行将函数 `greater9` 定义为在传入的数值大于 9 时返回 `true`, 否则返回 `false`。每个大于 9 的值换成新值 100。前两个迭代器参数应为正向迭代器, 使这个算法能修改序列中的元素。

第 53 行和第 54 行:

```
replace_copy_if( v4.begin(), v4.end(), c2.begin(),
                 greater9, 100 );
```

用函数 `replace_copy_if` 复制 vector `v4` 中从 `v4.begin()` 到 `v4.end()` (不包括 `v4.end()`) 的范围内所有元素。一元判定函数 `greater9` 返回 `true` 的元素换为新值 100。元素放在 vector `c2` 中从 `c2.begin()` 开始处。前两个迭代器参数应为输入迭代器, 第三个参数提供的迭代器应为输出迭代器, 使复制的元素能够插入复制位置。这个函数返回复制到 vector `c2` 的最后一个元素后面一位迭代器。

## 20.5.5 数学算法

图 20.30 演示标准库常用数学算法, 包括 `random_shuffle`、`count`、`count_if`、`min_element`、`max_element`、`accumulate`、`for_each` 和 `transform`。

```
1 // Fig. 20.30: fig20_30.cpp
```

```
2 // Examples of mathematical algorithms in the Standard Library.
3 #include <iostream>
4 #include <algorithm>
5 #include <numeric>      // accumulate is defined here
6 #include <vector>
7
8 using namespace std;
9
10 bool greater9( int );
11 void outputSquare( int );
12 int calculateCube( int );
13
14 int main()
15 {
16     const int SIZE = 10;
17     int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
18     vector< int > v( a1, a1 + SIZE );
19     ostream_iterator< int > output( cout, " " );
20
21     cout << "Vector v before random_shuffle: ";
22     copy( v.begin(), v.end(), output );
23     random_shuffle( v.begin(), v.end() );
24     cout << "\nVector v after random_shuffle: ";
25     copy( v.begin(), v.end(), output );
26
27     int a2[] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
28     vector< int > v2( a2, a2 + SIZE );
29     cout << "\n\nVector v2 contains: ";
30     copy( v2.begin(), v2.end(), output );
31     int result = count( v2.begin(), v2.end(), 8 );
32     cout << "\nNumber of elements matching 8: " << result;
33
34     result = count_if( v2.begin(), v2.end(), greater9 );
35     cout << "\nNumber of elements greater than 9: " << result;
36
37     cout << "\n\nMinimum element in Vector v2 is: "
38         << *( min_element( v2.begin(), v2.end() ) );
39
40     cout << "\n\nMaximum element in Vector v2 is: "
41         << *( max_element( v2.begin(), v2.end() ) );
42
43     cout << "\n\nThe total of the elements in Vector v is: "
44         << accumulate( v.begin(), v.end(), 0 );
45
46     cout << "\n\nThe square of every integer in Vector v is:\n";
47     for_each( v.begin(), v.end(), outputSquare );
48
49     vector< int > cubes( SIZE );
50     transform( v.begin(), v.end(), cubes.begin(),
51               calculateCube );
52     cout << "\n\nThe cube of every integer in Vector v is:\n";
53     copy( cubes.begin(), cubes.end(), output );
54
55     cout << endl;
56     return 0;
```

```

57 }
58
59 bool greater9( int value ) { return value > 9; }
60
61 void outputSquare( int value ) { cout << value * value << ' '; }
62
63 int calculateCube( int value ) { return value * value * value; }

```

**输出结果:**

```

Vector v before random_shuffle: 1 2 3 4 5 6 7 8 9 10
Vector v after random_shuffle: 5 4 1 3 7 8 9 10 6 2

```

```

Vector v2 contains: 100 2 8 1 50 3 8 8 9 10
Number of elements matching 8: 3
Number of elements greater than 9: 3

```

```

Minimum element in Vector v2 is: 1
Minimum element in Vector v2 is: 100

```

```

The total of the elements in Vector v is: 55

```

```

The square of every integer in Vector v is:
25 16 1 9 49 64 81 100 36 4

```

```

the cube of every integer in Vector v is:
125 64 1 27 343 512 729 1000 216 8

```

图 20.30 演示标准库常用数学算法

第 23 行:

```
random_shuffle( v.begin(), v.end() );
```

用函数 `random_shuffle` 随机排序 vector `v` 中从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 的元素。这个函数取两个随机访问迭代器参数。

第 31 行:

```
int result = count( v2.begin(), v2.end(), 8 );
```

用函数 `count` 计算 vector `v2` 中从 `v2.begin()` 到 `v2.end()` (但不包括 `v2.end()`) 的元素中数值为 8 的元素个数。这个函数要求两个迭代器参数至少为输入迭代器。

第 34 行:

```
result = count_if( v2.begin(), v2.end(), greater9 );
```

用函数 `count_if` 计算 vector `v2` 中从 `v2.begin()` 到 `v2.end()` (但不包括 `v2.end()`) 的元素中, 判定函数 `greater9` 返回 `true` 的元素个数。函数 `count_if` 要求两个迭代器参数至少为输入迭代器。

第 37 行和第 38 行:

```
cout << "\n\nMinimum element in Vector v2 is: "
    << *( min_element( v2.begin(), v2.end() ) );
```

用函数 `min_element` 计算 `vector v2` 中从 `v2.begin()` 到 `v2.end()` (但不包括 `v2.end()`) 的元素中的最小元素。函数返回最小元素位置的输入迭代器, 如果范围是空的, 则返回迭代器本身。函数要求两个迭代器参数至少为输入迭代器。这个函数的第二个版本取一个二元函数作为第三个参数, 比较序列中的元素。这个二元函数取两个参数并返回一个 `bool` 值。

#### 编程技巧 20.2

最好检查调用 `min_element` 中指定的范围不是空的, 或检查返回值不是“界外”迭代器。

第 40 行和第 41 行:

```
cout << "\nMaximum element in Vector v2 is: "
      << * ( max_element( v2.begin(), v2.end() ) );
```

用函数 `max_element` 计算 `vector v2` 中从 `v2.begin()` 到 `v2.end()` (但不包括 `v2.end()`) 的元素中的最大元素。函数返回最大元素位置的输入迭代器。如果范围是空的, 则返回迭代器本身。函数要求两个迭代器参数至少为输入迭代器。这个函数的第二个版本取第一个二元函数作为第三个参数, 比较序列中的元素。这个二元函数取两个参数并返回一个 `bool` 值。

第 43 行和第 44 行:

```
cout << "\n\nThe total of the elements in Vector v is: "
      << accumulate( v.begin(), v.end(), 0 );
```

用函数 `accumulate` (其原型在头文件 `<numeric>` 中) 求 `vector v` 中从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 的数值之和。这个函数的第二个版本用一个常用函数作为第二个参数, 确定元素如何累加求和。该常用函数取两个参数并返回一个结果, 第一个参数是累加结果的当前值, 第二个参数是序列中要累加的当前元素值。例如, 要累加每个元素的平方和, 可以用下列函数:

```
int sumOfSquares( int accumulator, int currentValue )
{
    return accumulator + currentValue * curentValue;
}
```

第 47 行:

```
for_each( v.begin(), v.end(), outputSquare );
```

用函数 `for_each` 对 `vector v` 中从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 的每个元素采用常用函数。这个常用函数取当前元素为参数, 且不修改这个元素。函数 `for_each` 要求其两个迭代器参数至少为输入迭代器。

第 50 行和第 51 行:

```
transform( v.begin(), v.end(), cubes.begin(),
           calculateCube );
```

用函数 `transform` 对 `vector v` 中从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 的每个元素采用常用函数。这个常用函数 (第 4 个参数) 取当前元素为参数, 且不修改这个元素, 返回经过 `transform` 操作的值。函数 `transform` 要求其两个迭代器参数至少为输入迭代器, 第三个参数至少为输出迭代器。第三个参数指定变换结果值的存放位置。注意, 第三个参数可以等于第一个参数。

## 20.5.6 基本查找与排序算法

图 20.31 演示标准库基本查找与排序算法，包括 find、find\_if、sort 和 binary\_search。

```
1 // Fig. 20.31: fig20_31.cpp
2 // Demonstrates search and sort capabilities.
3 #include <iostream>
4 #include <algorithm>
5 #include <vector>
6
7 using namespace std;
8
9 bool greater10( int value );
10
11 int main()
12 {
13     const int SIZE = 10;
14     int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
15     vector< int > v( a, a + SIZE );
16     ostream_iterator< int > output( cout, " " );
17
18     cout << "Vector v contains: ";
19     copy( v.begin(), v.end(), output );
20
21     vector< int >::iterator location;
22     location = find( v.begin(), v.end(), 16 );
23
24     if ( location != v.end() )
25         cout << "\n\nFound 16 at location "
26             << ( location - v.begin() );
27     else
28         cout << "\n\n16 not found";
29
30     location = find( v.begin(), v.end(), 100 );
31
32     if ( location != v.end() )
33         cout << "\n\nFound 100 at location "
34             << ( location - v.begin() );
35     else
36         cout << "\n\n100 not found";
37
38     location = find_if( v.begin(), v.end(), greater10 );
39
40     if ( location != v.end() )
41         cout << "\n\nThe first value greater than 10 is "
42             << *location << "\n\nfound at location "
43             << ( location - v.begin() );
44     else
45         cout << "\n\nNo values greater than 10 were found";
46
47     sort( v.begin(), v.end() );
48     cout << "\n\nVector v after sort: ";
49     copy( v.begin(), v.end(), output );
50
51     if ( binary_search( v.begin(), v.end(), 13 ) )
```

```

52     cout << "\n\n13 was found in v";
53     else
54         cout << "\n\n13 was not found in v";
55
56     if ( binary_search( v.begin(), v.end(), 100 ) )
57         cout << "\n100 was found in v";
58     else
59         cout << "\n100 was not found in v";
60
61     cout << endl;
62     return 0;
63 }
64
65 bool greater10( int value ) { return value > 10; }

```

**输出结果:**

Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4  
100 not found

The first value greater than 10 is 17  
found at location 2

vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v  
100 was not found in v

图 20.31 演示标准库基本查找与排序算法

**第22行:**

```
location = find( v.begin(), v.end(), 16 );
```

用函数 `find` 寻找 vector `v` 中从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 的元素中数值为 16 的元素。这个函数要求其两个迭代器参数至少为输入迭代器, 函数返回一个输入迭代器, 表示包含该值的第一个元素位置或表示序列结尾的迭代器。

**第38行:**

```
location = find_if( v.begin(), v.end(), greater10 );
```

用函数 `find_if` 寻找 vector `v` 中从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 的元素中, 一元判定函数 `greater10` 返回 `true` 的第一个值。函数 `greater10` 在第 65 行定义为取一个整数, 在整数参数大于 10 时返回 `true`, 否则返回 `false`。函数 `find_if` 要求其两个迭代器参数至少为输入迭代器, 该函数返回一个输入迭代器, 表示包含判定函数返回 `true` 的值的第一个元素位置或表示序列结尾的迭代器。

**第47行:**

```
sort( v.begin(), v.end() );
```

用函数 `sort` 按升序排列 vector `v` 中从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 的元素。这个函数要求其两个迭代器参数为随机访问迭代器。这个函数的第二个版本取一个二元判定函数作为第三个参数,

该判定函数取序列中的两个值作为参数，并返回一个表示排列顺序的 bool 值，如果两个元素已经符合排列顺序，则返回 true。

#### 常见编程错误 20.5

试图用随机访问迭代器之外的迭代器排序容器是个语法错误。函数 sort 要求随机访问迭代器。

第 51 行:

```
if ( binary_search( v.begin(), v.end(), 13 ) )
```

用函数 binary\_search 确定数值 13 是在 vector v 中从 v.begin() 到 v.end() (但不包括 v.end()) 的数值之中。函数 binary\_search 要求其两个迭代器参数至少为正向迭代器。该函数返回一个 bool 值，表示所要值是否在该序列中。这个函数的第二个版本取一个二元判定函数作为第四个参数，该判定函数取序列中的两个值作为参数，并返回一个表示排列顺序的 bool 值，如果两个元素已经符合排列顺序，则返回 true。

### 20.5.7 swap、iter\_swap 和 swap\_ranges

图 20.32 演示交换元素的 swap、iter\_swap 和 swap\_ranges 函数。

```
1 // Fig. 20.32: fig20_32.cpp
2 // Demonstrates iter_swap, swap and swap_ranges.
3 #include <iostream>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10     const int SIZE = 10;
11     int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
12     ostream_iterator< int > output( cout, " " );
13
14     cout << "Array a contains:\n";
15     copy( a, a + SIZE, output );
16
17     swap( a[ 0 ], a[ 1 ] );
18     cout << "\nArray a after swapping a[ 0 ] and a[ 1 ] "
19         << "using swap:\n";
20     copy( a, a + SIZE, output );
21
22     iter_swap( &a[ 0 ], &a[ 1 ] );
23     cout << "\nArray a after swapping a[ 0 ] and a[ 1 ] "
24         << "using iter_swap:\n";
25     copy( a, a + SIZE, output );
26
27     swap_ranges( a, a + 5, a + 5 );
28     cout << "\nArray a after swappiing the first five elements\n"
29         << "with the last five elements:\n";
30     copy( a, a + SIZE, output );
31
32     cout << endl;
33     return 0;
```



```
34 }
```

**输出结果:**

```
Array a contains:
1 2 3 4 5 6 7 8 9 10
Array a after swapping a[ 0] and a[ 1] using swap:
2 1 3 4 5 6 7 8 9 10
Array a after swapping a[ 0] and a [ 1] using iter_swap:
1 2 3 4 5 6 7 8 9 10
Array a after swapping the first five elements
with the last five elements:
6 7 8 9 10 1 2 3 4 5
```

图 20.32 演示交换元素的 swap、iter\_swap 和 swap\_ranges 函数

第 17 行:

```
swap( a[ 0 ], a[ 1 ] );
```

用函数 swap 交换两个值。本例中, 交换数组 a 的第一个元素和第二个元素。函数取要交换的两个值为参数。

第 22 行:

```
iter_swap( &a[ 0 ], &a[ 1 ] );
```

用函数 iter\_swap 交换两个值。函数取两个正向迭代器参数 (这里是数组元素的指针), 交换迭代器所指元素的值。

第 27 行:

```
swap_ranges( a, a + 5, a + 5 );
```

用函数 swap\_ranges 交换从 a 开始到 a+5 (但不包括 a+5) 的元素与从 a+5 开始的元素。函数取三个正向迭代器参数。第一个参数和第二个参数指定要交换的第一个序列元素范围, 第三个参数指定第二个序列开始元素的迭代器。本例中, 两个序列的值在相同数组中, 但也可以在不同数组或不同容器中。

## 20.5.8 copy\_backward、merge、unique 和 reverse

图 20.33 演示标准库函数 copy\_backward、merge、unique 和 reverse

```
1 // Fig. 20.33: fig20_33.cpp
2 // Demonstrates miscellaneous functions: copy_backward, merge,
3 // unique and reverse.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 int main()
11 {
12     const int SIZE = 5;
13     int a1[ SIZE ] = { 1, 3, 5, 7, 9 };
```

```

14  int a2[ SIZE ] = { 2, 4, 5, 7, 9 };
15  vector< int > v1( a1, a1 + SIZE );
16  vector< int > v2( a2, a2 + SIZE );
17
18  ostream_iterator< int > output( cout, " " );
19
20  cout << "Vector v1 contains: ";
21  copy( v1.begin(), v1.end(), output );
22  cout << "\nVector v2 contains: ";
23  copy( v2.begin(), v2.end(), output );
24
25  vector< int > results( v1.size() );
26  copy_backward( v1.begin(), v1.end(), results.end() );
27  cout << "\n\nAfter copy_backward, results contains: ";
28  copy( results.begin(), results.end(), output );
29
30  vector< int > results2( v1.size() + v2.size() );
31  merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
32        results2.begin() );
33  cout << "\n\nAfter merge of v1 and v2 results2 contains:\n";
34  copy( results2.begin(), results2.end(), output );
35
36  vector< int >::iterator endLocation;
37  endLocation = unique( results2.begin(), results2.end() );
38  cout << "\n\nAfter unique results2 contains:\n";
39  copy( results2.begin(), endLocation, output );
40
41  cout << "\n\nVector v1 after reverse: ";
42  reverse( v1.begin(), v1.end() );
43  copy( v1.begin(), v1.end(), output );
44
45  cout << endl;
46  return 0;
47 }

```

#### 输出结果:

Vector v1 contains: 1 3 5 7 9  
Vector v2 contains: 2 4 5 7 9

After copy\_backward results contains: 1 3 5 7 9

After merge of v1 and v2 results2 contains:  
1 2 3 4 5 5 7 7 9 9

After unique results2 contains:  
1 2 3 4 5 7 9

Vector v1 after reverse: 9 7 5 3 1

图 20.33 演示标准库函数 `copy_backward`、`merge`、`unique` 和 `reverse`

#### 第26行:

```
copy_backward( v1.begin(), v1.end(), results.end() );
```

用函数 `copy_backward` 复制 `vector v1` 中从 `v1.begin()` 到 `v1.end()` (但不包括 `v1.end()`) 的元素, 并将这些元素放在 `vector results` 中从 `results.end()` 开始的位置, 并由此向 `vector` 开头复制。函数返回复制到 `vector results` 的最后一个元素位置的迭代器 (即 `results` 开头, 因为我们用逆向复制)。元素按与 `v1` 的相同顺序放在 `results` 中。这个函数要求三个双向迭代器参数 (可以通过序列正向或逆向递增和递减迭代器)。 `copy` 和 `copy_backward` 之间的主要差别在于 `copy` 返回的迭代器放在所复制的最后一个元素后面, 而 `copy_backward` 返回复制的最后一个元素位置的迭代器 (即序列中第一个元素)。另外, `copy` 要求两个输入迭代器和一个输出迭代器参数。

第31行和第32行:

```
merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
       results2.begin() );
```

用函数 `merge` 组合两个升序序列, 变为第三个升序序列。函数要求五个迭代器参数。前4个迭代器至少应为输入迭代器, 最后一个参数至少应为输出迭代器。前两个参数指定第一个排序序列 (`v1`) 中的元素范围, 接着的两个参数指定第二个排序序列 (`v2`) 中的元素范围, 最后一个参数指定第三个序列 (`results2`) 中开始合并元素的位置。这个函数第二个版本中的第五个参数是指定排列顺序的二元判定函数。

注意第30行生成的 `vector` 得到 `v1.size() + v2.size()` 个元素。这样使用 `merge` 函数时要求存放结果的序列长度至少应为两个合并序列长度之和。如果在 `merge` 操作之前不对结果序列分配足够的长度, 则可以用下列语句:

```
vector< int > results2();
merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
       back_inserter( results2 ) );
```

参数 `back_inserter( results2 )` 对容器 `results2` 使用函数模板 `back_inserter` (在头文件 `<iterator>` 中)。 `back_inserter` 调用容器的默认 `push_back` 函数, 在容器末尾插入一个元素。更重要的是, 如果要插入元素的容器中没有更多可用元素, 则容器自动增加长度。这样, 容器中的元素个数事先不必知道。还有第二种插入器 `front_inserter` (在参数指定的容器开头插入元素) 和 `inserter` (在第一个参数指定的容器中第二个参数指定的迭代器之前插入元素)。

第37行:

```
endLocation = unique( results2.begin(), results2.end() );
```

用函数 `unique` 处理 `vector results2` 中从 `results2.begin()` 到 `results2.end()` (但不包括 `results2.end()`) 的已排序元素序列。对具有重复值的排序元素序列使用这个函数之后, 每个值只在序列中保持单个副本。该函数取两个参数 (至少应为正向迭代器), 返回的迭代器指向惟一值序列中最后一个元素后面一位。容器中最后一个惟一值后面的所有元素都是未定义的。这个函数的第二个版本取一个二元判定函数作为第三个参数, 指定如何比较两个元素的相等性。

第42行:

```
reverse( v1.begin(), v1.end() );
```

用函数 `reverse` 逆转 `vector v1` 中从 `v1.begin()` 到 `v1.end()` (但不包括 `v1.end()`) 的所有元素。函数取两个参数, 这两个参数至少应为双向迭代器。

### 20.5.9 inplace\_merge、unique\_copy 和 reverse\_copy

图 20.34 的程序演示标准库函数 `inplace_merge`、`unique_copy` 和 `reverse_copy`。

```
1 // Fig. 20.34: fig20_34.cpp
2 // Demonstrates miscellaneous functions: inplace_merge,
3 // reverse_copy, and unique_copy.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7 #include <iterator>
8
9 using namespace std;
10
11 int main()
12 {
13     const int SIZE = 10;
14     int a1[ SIZE ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
15     vector< int > v1( a1, a1 + SIZE );
16
17     ostream_iterator< int > output( cout, " " );
18
19     cout << "Vector v1 contains: ";
20     copy( v1.begin(), v1.end(), output );
21
22     inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
23     cout << "\nAfter inplace_merge, v1 contains: ";
24     copy( v1.begin(), v1.end(), output );
25
26     vector< int > results1;
27     unique_copy( v1.begin(), v1.end(),
28                 back_inserter( results1 ) )
29     cout << "\nAfter unique_copy results1 contains: ";
30     copy( results1.begin(), results1.end(), output );
31
32     vector< int > results2;
33     cout << "\nAfter reverse_copy, results2 contains: ";
34     reverse_copy( v1.begin(), v1.end(),
35                  back_inserter( results2 ) );
36     copy( results2.begin(), results2.end(), output );
37
38     cout << endl;
39     return 0;
40 }
```

**输出结果：**

```
Vector v1 contains: 1 3 5 7 9 1 3 5 7 9
After inplace_merge, v1 contains: 1 1 3 3 5 5 7 7 9 9
After unique_copy results1 contains: 1 3 5 7 9
After reverse_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1
```

图 20.34 演示标准库函数 `inplace_merge`、`unique_copy` 和 `reverse_copy`

第22行:

```
inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
```

用函数 `inplace_merge` 合并同一容器中的两个排序元素序列。本例中, 从 `v1.begin()` 到 `v1.begin() + 5` (但不包括 `v1.begin() + 5`) 的元素合并到从 `v1.begin() + 5` 到 `v1.end()` (但不包括 `v1.end()`) 的元素中。这个函数取三个参数, 至少应为双向迭代器。这个函数的第二个版本取一个二元判定函数作为第四个参数, 指定如何比较序列中的两个元素。

第27行第28行:

```
unique_copy( v1.begin(), v1.end(),
             back_inserter( results1 ) )
```

用函数 `unique_copy` 复制从 `v1.begin()` 到 `v1.end()` (但不包括 `v1.end()`) 的所有唯一元素。复制的元素放在 `results1` 中。前两个参数至少应为输入迭代器, 第三个参数至少应为输出迭代器。本例中, 我们事先不在 `results1` 中分配足够空间存放从 `v1` 复制的所有元素, 而是让函数 `back_inserter` (在头文件 `<iterator>` 中定义) 将元素加到 `vector v1` 末尾。`back_inserter` 使用 `vector` 类的功能在 `vector` 末尾插入元素。由于 `back_inserter` 插入元素而不替换现有元素值, 因此 `vector` 可以根据需要增加元素。`unique_copy` 函数的第二个版本取一个二元判定函数作为第四个参数, 指定如何比较序列中两个元素的相等性。

第34行和35行:

```
reverse_copy( v1.begin(), v1.end(),
              back_inserter( results2 ) );
```

用函数 `reverse_copy` 逆向复制从 `v1.begin()` 到 `v1.end()` (但不包括 `v1.end()`) 的所有元素。复制的元素放在 `vector results2` 中, 用 `back_inserter` 对象保证 `vector` 能在需要时增加元素。`reverse_copy` 的前两个参数至少应为双向迭代器, 第三个参数至少应为输出迭代器。

### 20.5.10 集合操作

图 20.35 演示标准库函数 `includes`、`set_difference`、`set_intersection`、`set_symmetric_difference` 和 `set_union`, 用于操作排序值的集合。为了演示标准库函数适用于数组和容器, 本例只用数组 (记住, 数组指针是随机访问迭代器)。

```
1 // Fig. 20.35: fig20_35.cpp
2 // Demonstrates includes, set_difference, set_intersection,
3 // set_symmetric_difference and set_union.
4 #include <iostream>
5 #include <algorithm>
6
7 using namespace std;
8
9 int main()
10 {
11     const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
12     int a1[ SIZE1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13     int a2[ SIZE2 ] = { 4, 5, 6, 7, 8 };
14     int a3[ SIZE3 ] = { 4, 5, 6, 11, 15 };
```

```

15 ostream_iterator< int > output( cout, " " );
16
17 cout << "a1 contains: ";
18 copy( a1, a1 + SIZE1, output );
19 cout << "\na2 contains: ";
20 copy( a2, a2 + SIZE2, output );
21 cout << "\na3 contains: ";
22 copy( a3, a3 + SIZE2, output );
23
24 if ( includes( a1, a1 + SIZE1, a2, a2 + SIZE2 ) )
25     cout << "\na1 includes a2";
26 else
27     cout << "\na1 does not include a2";
28
29 if ( includes( a1, a1 + SIZE1, a3, a3 + SIZE2 ) )
30     cout << "\na1 includes a3";
31 else
32     cout << "\na1 does not include a3";
33
34 int difference[ SIZE1 ];
35 int *ptr = set_difference( a1, a1 + SIZE1, a2, a2 + SIZE2,
36                           difference );
37 cout << "\nset_difference of a1 and a2 is: ";
38 copy( difference, ptr, output );
39
40 int intersection[ SIZE1 ];
41 ptr = set_intersection( a1, a1 + SIZE1, a2, a2 + SIZE2,
42                          intersection );
43 cout << "\nset_intersection of a1 and a2 is: ";
44 copy( intersection, ptr, output );
45
46 int symmetric_difference[ SIZE1 ];
47 ptr = set_symmetric_difference( a1, a1 + SIZE1,
48                                a2, a2 + SIZE2, symmetric_difference );
49 cout << "\nset_symmetric_difference of a1 and a2 is: ";
50 copy( symmetric_difference, ptr, output );
51
52 int unionSet[ SIZE3 ];
53 ptr = set_union( a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet );
54 cout << "\nset_union of a1 and a3 is: ";
55 copy( unionSet, ptr, output );
56 cout << endl;
57 return 0;
58 }

```

**输出结果:**

```

a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15
a1 includes a2
a1 does not include a3
set_difference of a1 and a2 is: 1 2 3 9 10
set_intersection of a1 and a2 is: 4 5 6 7 8
set_symmetric_difference of a1 and a2 is: 1 2 3 9 10
set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15

```

图 20.35 演示标准库的 set 操作

第24行:

```
if ( includes( a1, a1 + SIZE1, a2, a2 + SIZE2 ) )
```

在if结构条件中调用函数includes。函数includes比较两组排序值,确定第二个集合的每个元素是否在第一个集合中。如果是,则includes返回true,否则includes返回false。前两个参数至少应为输入迭代器,描述第一组值集合。本例中,第一个集合包括从a1到a1 + SIZE1(但不包括a1 + SIZE1)的元素。后两个参数至少应为输入迭代器,描述第二组值集合。本例中,第二个集合包括从a2到a2 + SIZE2(但不包括a2 + SIZE2)的元素。这个函数的第二个版本取一个二元判定函数作为第五个参数,指定如何比较两个元素的相等性。

第35行和第36行:

```
int *ptr = set_difference( a1, a1 + SIZE1, a2, a2 + SIZE2,  
                           difference );
```

用函数set\_difference确定第一个排序值集合中不在第二个排序值集合中的元素(两个集合都应按升序排列)。差值元素复制到第五个参数中(这里是数组difference)。前两个参数至少应为输入迭代器,描述第一组值集合。接着的两个参数至少应为输入迭代器,描述第二组值集合。第五个参数至少应为输出迭代器,表示差值复制到什么位置。函数返回一个输出迭代器,放在第五个参数所指集合中最后复制值的后面一位。set\_difference函数的第二个版本取一个二元判定函数作为第六个参数,表示元素最初排列顺序。两个序列要用相同的比较函数排序。

第41行和第42行:

```
ptr = set_intersection( a1, a1 + SIZE1, a2, a2 + SIZE2,  
                        intersection );
```

用函数set\_intersection确定第一个排序元素集合与第二个排序元素集合的交集(两个集合都应按升序排列)。两个集合共有的元素复制到第五个参数中(这里是数组intersection)。前两个参数至少应为输入迭代器,描述第一组值集合。接着的两个参数至少应为输入迭代器,描述第二组值集合。第五个参数至少应为输出迭代器,表示相交值复制到什么位置。函数返回一个输出迭代器,放在第五个参数所指集合中最后复制值的后面一位。set\_intersection函数的第二个版本取一个二元判定函数作为第六个参数,表示元素最初排列顺序。两个序列要用相同的比较函数排序。

第47行和48行:

```
ptr = set_symmetric_difference( a1, a1 + SIZE1,  
                                a2, a2 + SIZE2, symmetric_difference );
```

用函数set\_symmetric\_difference确定第一个集合中不在第二个集合中的元素和第二个集合中不在第一个集合中的元素(两个集合都应按升序排列)。两个集合的差值元素复制到第五个参数中(这里是数组symmetric\_difference)。前两个参数至少应为输入迭代器,描述第一组值集合。接着的两个参数至少应为输入迭代器,描述第二组值集合。第五个参数至少应为输出迭代器,表示差值复制到什么位置。函数返回一个输出迭代器,放在第五个参数所指集合中最后复制值的后面一位。set\_symmetric\_difference函数的第二个版本取一个二元判定函数作为第六个参数,表示元素最初排列顺序。两个序列要用相同的比较函数排序。

第53行:

```
ptr = set_union( a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet );
```

用函数 `set_union` 确定第一个排序元素集合与第二个排序元素集合的并集（两个集合都应按升序排列）。两个集合的元素复制到第五个参数中（这里是数组 `unionSet`）。前两个参数至少应为输入迭代器，描述第一组值集合。接着的两个参数至少应为输入迭代器，描述第二组值集合。第五个参数至少应为输出迭代器，表示相并值复制到什么位置。函数返回一个输出迭代器，放在第五个参数所指集合中最后复制值的后面一位。`set_union` 函数的第二个版本取一个二元判定函数作为第六个参数，表示元素最初排列顺序。两个序列要用相同的比较函数排序。

### 20.5.11 lower\_bound、upper\_bound 和 equal\_range

图 20.36 演示标准库函数 `lower_bound`、`upper_bound` 和 `equal_range`。

```
1 // Fig. 20.36: fig20_36.cpp
2 // Demonstrates lower_bound, upper_bound and equal_range for
3 // a sorted sequence of values.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 int main()
11 {
12     const int SIZE = 10;
13     int a1[] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
14     vector< int > v( a1, a1 + SIZE );
15     ostream_iterator< int > output( cout, " " );
16
17     cout << "Vector v contains:\n";
18     copy( v.begin(), v.end(), output );
19
20     vector< int >::iterator lower;
21     lower = lower_bound( v.begin(), v.end(), 6 );
22     cout << "\n\nLower bound of 6 is element "
23          << ( lower - v.begin() ) << " of vector v";
24
25     vector< int >::iterator upper;
26     upper = upper_bound( v.begin(), v.end(), 6 );
27     cout << "\n\nUpper bound of 6 is element "
28          << ( upper - v.begin() ) << " of vector v";
29
30     pair< vector< int >::iterator, vector< int >::iterator > eq;
31     eq = equal_range( v.begin(), v.end(), 6 );
32     cout << "\n\nUsing equal_range:\n"
33          << "    Lower bound of 6 is element "
34          << ( eq.first - v.begin() ) << " of vector v";
35     cout << "\n    Upper bound of 6 is element "
36          << ( eq.second - v.begin() ) << " of vector v";
37
38     cout << "\n\nUse lower_bound to locate the first point\n"
39          << "at which 5 can be inserted in order";
40     lower = lower_bound( v.begin(), v.end(), 5 );
```



```

41  cout << "\n  Lower bound of 5 is element "
42      << ( lower - v.begin() ) << " of vector v";
43
44  cout << "\n\nUse upper_bound to locate the last point\n"
45      << "at which 7 can be inserted in order";
46  upper = upper_bound( v.begin(), v.end(), 7 );
47  cout << "\n  Upper bound of 7 is element "
48      << ( upper - v.begin() ) << " of vector v";
49
50  cout << "\n\nUse equal_ranng to locate the first and\n"
51      << "last point at which 5 can be inserted in order";
52  eq = equal_range( v.begin(), v.end(), 5 );
53  cout << "\n  Lower bound of 5 is element "
54      << ( eq.first - v.begin() ) << " of vector v";
55  cout << "\n  Upper bound of 5 is element "
56      << ( eq.second - v.begin() ) << " of vector v"
57      << endl;
58  return 0;
59 }

```

**输出结果:**

Vector v contains:  
2 2 4 4 4 6 6 6 6 8

Lower bound of 6 is element 5 of vector v  
Upper bound of 6 is element 9 of vector v  
Using equal\_range:  
Lower bound of 6 is element 5 of vector v  
Upper bound of 6 is element 9 of vector v

Use lower\_bound to locate the first point  
at which 5 can be inserted in order  
Lower bound of 5 is element 5 of vector v

Use lower\_bound to locate the first point  
at which 7 can be inserted in order  
Upper bound of 7 is element 9 of vector v

Use equal\_range to locate the first and  
last point at which 5 can be inserted in order  
Lower bound of 5 is element 5 of vector v  
Upper bound of 5 is element 5 of vector v

图 20.36 演示标准库函数 lower\_bound、upper\_bound 和 equal\_range

**第21行:**

```
lower = lower_bound( v.begin(), v.end(), 6 );
```

用函数 lower\_bound 确定排序数值序列中插入第三个参数而保持升序的第一个位置。前两个迭代器参数至少应为正向迭代器。第三个参数是确定下限的值。函数返回一个正向迭代器, 指向插入的位置。lower\_bound 函数的第2个版本取一个二元判定函数作为第四个参数, 表示元素最初排列的顺序。

**第26行:**

```
upper = upper_bound( v.begin(), v.end(), 6 );
```

用函数 `upper_bound` 确定排序数值序列中插入第三个参数而保持升序的最后一个位置。前两个迭代器参数至少应为正向迭代器。第三个参数是确定上限的值。函数返回一个正向迭代器，指向插入的位置。`upper_bound` 函数的第2个版本取一个二元判定函数作为第四个参数，表示元素最初排列的顺序。

第31行：

```
eq = equal_range( v.begin(), v.end(), 6 );
```

用函数 `equal_range` 返回一个正向迭代器对，包含 `lower_bound` 和 `upper_bound` 操作的组合结果。前两个迭代器参数至少应为正向迭代器。第三个参数是确定相等范围的值。函数分别返回下限 (`eq.first`) 和上限 (`eq.second`) 的正向迭代器对。

函数 `lower_bound`、`upper_bound` 和 `equal_range` 常用于寻找排序序列中的插入点。第40行用 `lower_bound` 寻找 `v` 中可以顺序插入数值5的第一个点。第46行用 `upper_bound` 寻找 `v` 中可以顺序插入数值7的最后一个点。第52行用 `equal_range` 寻找 `v` 中可以插入数值5的第一个和最后一个点。

## 20.5.12 堆排序

图20.37演示标准库函数中进行堆排序的排序算法。堆排序算法将元素数组排列成特殊的二叉树，称为堆 (heap)。堆的关键特性是最大元素总是在堆的顶上，二叉树中任何节点的子节点值总是小于或等于该节点的值。这种堆布置称为最大堆 (maxheap)。堆排序算法通常在“数据结构”和“算法”课题中介绍。

```
1 // Fig. 20.37: fig20_37.cpp
2 // Demonstrating push_heap, pop_heap, make_heap and sort_heap.
3 #include <iostream>
4 #include <algorithm>
5 #include <vector>
6
7 using namespace std;
8
9 int main()
10 {
11     const int SIZE = 10;
12     int a[ SIZE ] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
13     int i;
14     vector< int > v( a, a + SIZE ), v2;
15     ostream_iterator< int > output( cout, " " );
16
17     cout << "Vector v before make_heap:\n";
18     copy( v.begin(), v.end(), output );
19     make_heap( v.begin(), v.end() );
20     cout << "\nVector v after make_heap:\n";
21     copy( v.begin(), v.end(), output );
22     sort_heap( v.begin(), v.end() );
23     cout << "\nVector v after sort_heap:\n";
24     copy( v.begin(), v.end(), output );
25
26     // perform the heapsort with push_heap and pop_heap
27     cout << "\n\nArray a contains: ";
```

```

28     copy( a, a + SIZE, output );
29
30     for ( i = 0; i < SIZE; ++i ) {
31         v2.push_back( a[ i ] );
32         push_heap( v2.begin(), v2.end() );
33         cout << "\nv2 after push_heap(a[" << i << "]): ";
34         copy( v2.begin(), v2.end(), output );
35     }
36
37     for ( i = 0; i < v2.size(); ++i ) {
38         cout << "\nv2 after " << v2[ 0 ] << " popped from heap\n";
39         pop_heap( v2.begin(), v2.end() - i );
40         copy( v2.begin(), v2.end(), output );
41     }
42
43     cout << endl;
44     return 0;
45 }

```

**输出结果:**

Vector v before make\_heap:

3 100 52 77 22 31 1 98 13 40

Vector v after make\_heap:

100 98 52 77 40 31 1 3 13 22

Vector v after sort\_heap:

1 3 13 22 31 40 52 77 98 100

Array a contains: 3 100 52 77 22 31 1 98 13 40

v2 after push\_heap(a[ 0 ]): 3

v2 after push\_heap(a[ 1 ]): 100 3

v2 after push\_heap(a[ 2 ]): 100 3 52

v2 after push\_heap(a[ 3 ]): 100 77 52 3

v2 after push\_heap(a[ 4 ]): 100 77 52 3 22

v2 after push\_heap(a[ 5 ]): 100 77 52 3 22 31

v2 after push\_heap(a[ 6 ]): 100 77 52 3 22 31 1

v2 after push\_heap(a[ 7 ]): 100 98 52 77 22 31 1 3

v2 after push\_heap(a[ 8 ]): 100 98 52 77 22 31 1 3 13

v2 after push\_heap(a[ 9 ]): 100 98 52 77 40 31 1 3 13 22

v2 after 100 popped from heap

98 77 52 22 40 31 1 3 13 100

v2 after 98 popped from heap

77 40 52 22 13 31 1 3 98 100

v2 after 77 popped from heap

52 40 31 22 13 3 1 77 98 100

v2 after 52 popped from heap

40 22 31 1 13 3 52 77 98 100

v2 after 40 popped from heap

31 22 3 1 13 40 52 77 98 100

v2 after 31 popped from heap

22 13 3 1 31 40 52 77 98 100

v2 after 22 popped from heap

13 1 3 22 31 40 52 77 98 100

v2 after 13 popped from heap

3 1 13 22 31 40 52 77 98 100

v2 after 3 popped from heap

```

1 3 13 22 31 40 52 77 98 100
v2 after 1 popped from heap
1 3 13 22 31 40 52 77 98 100

```

图 20.37 演示标准库函数中进行堆排序的函数

第 19 行:

```
make_heap( v.begin(), v.end() );
```

用函数 `make_heap` 取得从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 范围的数值序列, 并生成一个堆, 用于产生排序序列。两个迭代器参数应为随机访问迭代器, 因此这个函数只适用于数组、`vector` 和 `deque`。这个函数的第二个版本取第三个参数, 是比较数值的二元判定函数。

第 22 行:

```
sort_heap( v.begin(), v.end() );
```

用 `sort_heap` 函数排序从 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 范围的数值序列。两个迭代器参数应为随机访问迭代器。这个函数的第二个版本取第三个参数, 是比较数值的二元判定函数。

第 32 行:

```
push_heap( v2.begin(), v2.end() );
```

用函数 `push_heap` 在堆中增加新值。我们一次取数组 `a` 中的一个元素, 将这个元素添加到 `vector v2` 的末尾, 并进行 `push_heap` 操作。如果添加的元素是 `vector` 中的惟一元素, 则 `vector` 已经是堆。否则函数 `push_heap` 重新将 `vector` 元素排列成堆。每次调用 `push_heap` 时, 它假设当前在 `vector` 中的最后一个元素 (即调用 `push_heap` 之前添加的元素) 是要加进堆中的元素, `vector` 中的所有其他元素已经排列成堆。`push_heap` 的两个迭代器参数应为随机访问迭代器。这个函数的第二个版本取第三个参数, 是比较数值的二元判定函数。

第 39 行:

```
pop_heap( v2.begin(), v2.end() - i );
```

用函数 `pop_heap` 删除堆顶部的元素。这个函数假设两个随机访问迭代器参数指定范围中的元素已形成一个堆。重复删除堆顶部的元素最终会得到排序的数值序列。函数 `pop_heap` 交换堆中第一个元素 (这里是 `v2.begin()`) 和堆中最后一个元素 (`v2.end()-i` 之前的元素), 然后保证最后一个元素之前的元素仍然形成堆。注意 `pop_heap` 操作之后, 输出中的 `vector` 按升序排列。这个函数的第二个版本取第三个参数, 是比较数值的二元判定函数。

### 20.5.13 min 和 max

算法 `min` 和 `max` 分别确定两个元素中的最小值与最大值。图 20.38 的程序演示 `int` 和 `char` 值的 `min` 和 `max`。

```

1 // Fig. 20.38: fig20_38.cpp
2 // Demonstrating min and max
3 #include <iostream>
4 #include <algorithm>
5
6 using namespace std;

```

```

7
8 int main()
9 {
10     cout << "The minimum of 12 and 7 is: " << min( 12, 7 );
11     cout << "\nThe maximum of 12 and 7 is: " << max( 12, 7 );
12     cout << "\nThe minimum of iG* and iZ* is: "
13         << min( iG*, iZ* );
14     cout << "\nThe maximum of iG* and iZ* is: "
15         << max( iG*, iZ* ) << endl;
16     return 0;
17 }

```

**输出结果:**

```

The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
The minimum of 'G' and 'Z' is: G
The maximum of 'G' and 'Z' is: z

```

图 20.38 演示算法 min 和 max

## 20.5.14 本章未介绍的算法

图 20.39 列出本章未介绍的算法。

算法	说明
adjacent_difference	从序列中第二个元素开始, 计算当前元素与前一元素之间的差值 (用运算符 -), 并存放结果。前两个输入迭代器参数表示容器中的元素范围, 第三个输出迭代器参数表示存放结果的位置。这个函数的第二个版本取一个二元函数作为第四个参数, 进行当前元素与上一元素的计算
inner_product	这个函数计算两个序列的积之和。将每个序列中的对应元素相乘, 然后将结果求和
partial_sum	计算序列中数值的动态和 (用运算符 +)。前两个输入迭代器参数表示容器中的元素范围, 第三个输出迭代器参数表示存放结果的位置。这个函数的第二个版本取一个二元函数作为第四个参数, 进行当前元素值与动态和的计算
nth_element	这个函数用三个随机访问迭代器将元素范围进行划分。第一个和最后一个参数表示元素范围, 第二个参数是分区元素的位置。执行这个函数之后, 分区元素左边的所有元素小于分区元素, 分区元素右边的所有元素大于或等于分区元素。这个函数的第二个版本取一个二元比较函数作为第四个参数
partition	这个函数与 nth_element 相似, 但只要求功能更弱的双向迭代器即可, 因此比 nth_element 更加灵活。函数 partition 要求两个双向迭代器表示分区元素范围。第三个参数是一个一元判定函数, 用于分区所有元素, 使序列中判定为 true 的所有元素放在左边, 判定为 false 的所有元素放在右边。返回一个双向迭代器, 表示判定为 false 的序列中第一个元素
stable_partition	这个函数与 partition 相似, 但判定函数返回 true 的元素保持原有顺序, 判定函数返回 false 的元素也保持原有顺序
next_permutation	序列中的下一个词法置换
prev_permutation	序列中的前一个词法置换
rotate	这个函数取三个正向迭代器参数, 根据从第二个参数减去第一个参数得到的位置号, 将第一个和最后一个参数表示的序列移动相应位数。例如, 序列 1、2、3、4、5 移两位就变成 4、5、1、2、3
rotate_copy	这个函数与 rotate 相似, 只是结果存放在第四个输出迭代器参数指定的序列中。两个序列的元素个数应相同

(续表)

算法	说明
<code>adjacent_find</code>	这个函数返回一个输入迭代器,表示序列中第一个与相邻元素相同的元素。如果没有与相邻元素相同的元素,则迭代器指向序列末尾
<code>partial_sort</code>	这个函数用三个随机访问迭代器排序序列的一部分。第一个参数和最后一个参数表示整个元素序列。第二个参数表示排序部分的结束位置。默认情况下,元素用运算符<排序(也可以提供二元判定函数)。第二个迭代器参数到序列末尾的顺序是未定义的
<code>partial_sort_copy</code>	这个函数用两个输入迭代器和两个随机访问迭代器排序两个输入迭代器表示的序列的一部分。结果存放在用两个随机访问迭代器参数表示的序列中。默认情况下,元素用运算符<排序(也可以提供二元判定函数)。排序的元素个数是结果元素个数和原序列元素个数中的较小者
<code>stable_sort</code>	这个函数类似于 <code>sort</code> ,只是所有相等元素保持原有顺序

图 20.39 本章未介绍的算法

## 20.6 bitset 类

bitset 类使位集合更容易生成和操作。位集合是用于表示位标志的集合。bitset 在编译时是固定长度的。

```
bitset< size > b;
```

生成的 bitset b 中每个位均初始化为 0。

```
b.set( bitNumber );
```

将 bitset b 的 bitNumber 位设置为“开”,表达式 `b.set()` 将 b 中的所有位设置为“开”。

```
b.reset( bitNumber );
```

将 bitset b 的 bitNumber 位设置为“关”,表达式 `b.reset()` 将 b 中的所有位设置为“关”。

```
b.flip( bitNumber );
```

将 bitset b 的 bitNumber 位翻转(即如果位为开,则 flip 将其设置为关)。表达式 `b.flip()` 将 b 的所有位翻转。

```
b[ bitNumber ];
```

返回 bitset b 中 bitNumber 的引用。同样,

```
b.at( bitNumber );
```

首先对 bitNumber 进行范围检查,如果 bitNumber 在范围之内,则 at 返回该位的引用,否则 at 抛出 `out_of_range` 异常。

```
b.test( bitNumber );
```

首先对 bitNumber 进行范围检查,如果 bitNumber 在范围之内,则 test 在位打开时返回 true,位关掉时返回 false,否则 test 抛出 `out_of_range` 异常。

表达式:

```
b.size()
```

返回 bitset b 中的位数。

表达式:

```
b.count()
```

返回 bitset b 中设置的位数。

表达式:

```
b.any()
```

在 bitset b 中的位设置时返回 true。

表达式:

```
b.none()
```

在 bitset b 中的位没有设置时返回 true。

表达式:

```
b == b1  
b != b1
```

分别比较两个 bitset 的相等性与不等性。

可以用位赋值运算符 `&=`、`|=` 和 `^=` 组合 bitset。例如:

```
b &= b1;
```

进行 bitset b 与 b1 之间的位逻辑与, 结果存放在 b 中。位逻辑或和位逻辑异或操作如下:

```
b |= b1;  
b ^= b2;
```

• 下列表达式:

```
b >>= n;
```

将 bitset b 中的位右移 n 位。

下列表达式:

```
b <<= n;
```

将 bitset b 中的位左移 n 位。

表达式:

```
b.to_string()  
b.to_ulong()
```

分别将 bitset b 变为 string 和 long。

图 20.40 再次用 Eratosthenes 筛选法寻找质数 (见练习 4.29)。这里用 bitset 而不用数组实现算法。程序显示从 2 到 1023 的所有质数, 然后让用户输入一个数, 确定其是否质数。

```
1 // Fig. 20.40: fig20_40.cpp  
2 // Using a bitset to demonstrate the Sieve of Eratosthenes.  
3 #include <iostream>  
4 #include <iomanip>
```

```

5 #include <bitset>
6 #include <cmath>
7
8 using namespace std;
9
10 int main()
11 {
12     const int size = 1024;
13     int i, value, counter;
14     bitset< size > sieve;
15
16     sieve.flip();
17
18     // perform Sieve of Eratosthenes
19     int finalBit = sqrt( sieve.size() ) + 1;
20
21     for ( i = 2; i < finalBit; ++i )
22         if ( sieve.test( i ) )
23             for ( int j = 2 * i; j < size; j += i )
24                 sieve.reset( j );
25
26     cout << "The prime numbers in the range 2 to 1023 are:\n";
27
28     for ( i = 2, counter = 0; i < size; ++i )
29         if ( sieve.test( i ) ) {
30             cout << setw( 5 ) << i;
31
32             if ( ++counter % 12 == 0 )
33                 cout << '\n';
34         }
35
36     cout << endl;
37
38     // get a value from the user to determine if it is prime
39     cout << "\nEnter a value from 1 to 1023 (-1 to end): ";
40     cin >> value;
41
42     while ( value != -1 ) {
43         if ( sieve[ value ] )
44             cout << value << " is a prime number\n";
45         else
46             cout << value << " is not a prime number\n";
47
48         cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
49         cin >> value;
50     }
51
52     return 0;
53 }

```

**输出结果:**

The prime numbers in the range 2 to 1023 are:

```

2    3    5    7   11   13   17   19   23   29   31   37
41   43   47   53   59   61   67   71   73   79   83   89
97  101  103  107  109  113  127  131  137  139  149  151

```



```

157 163 167 173 179 181 191 193 197 199 211 223
227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359
367 373 379 383 389 397 401 409 419 421 431 433
439 443 449 457 461 463 467 479 487 491 499 503
509 521 523 541 547 557 563 569 571 577 587 593
599 601 607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733 739 743
751 757 761 769 773 787 797 809 811 821 823 827
829 839 853 857 859 863 877 881 883 887 907 911
919 929 937 941 947 953 967 971 977 983 991 997
1009 1013 1019 1021

```

```

Enter a value from 1 to 1023 (-1 to end): 389
389 is a prime number

```

```

Enter a value from 2 to 1023 (-1 to end): 88
88 is not a prime number

```

```

Enter a value from 2 to 1023 (-1 to end): -1

```

图 20.40 bitset 类与 Eratosthenes 筛选法

第 14 行:

```
bitset< size > sieve;
```

生成 size 位的 bitset (size 在本例中为 1024)。我们在这个程序中忽略第 0 位和第 1 位的位。默认情况下, bitset 中所有位均为关闭状态。下列代码:

```

// perform Sieve of Eratosthenes
int finalBit = sqrt( sieve.size() ) + 1;

for ( i = 2; i < finalBit; ++i )
    if ( sieve.test( i ) )
        for ( int j = 2 * i; j < size; j += i )
            sieve.reset( j );

```

确定 2 到 1023 的所有质数。整数 finalBit 确定算法何时完成。基本算法是, 如果一个数只能被 1 和本身整除, 则该数是个质数。从数字 2 开始, 确定一个数是质数之后, 就可以删除这个数的所有倍数。数字 2 只能被 1 和本身整除, 因此是个质数, 则可以删除 4、6、8 等等。数字 3 只能被 1 和本身整除, 因此是个质数, 则可以删除 3 的所有倍数 (记住, 所有偶数已经删除)。

## 20.7 函数对象

函数对象 (function object) 和函数适配器使 STL 更加灵活。函数对象包含可以用 operator() 调用的函数。STL 函数对象和适配器的原型在 <functional> 中。函数对象还可以用所在函数封装数据。标准函数对象是内联函数, 以提高性能。表 20.41 列出了标准库中的 STL 函数对象。

图 20.42 的程序演示用 accumulate 数字算法 (见图 20.30) 计算 vector 中元素的平方和。accumulate 的第 4 个参数是个二元函数对象或函数指针, 相应二元函数取两个参数并返回一个结果。函数 accumulate 声明两次, 一次用二元函数的函数指针, 一次用函数对象。

STL 函数对象	类型
<code>divides&lt;T&gt;</code>	算术
<code>equal_to&lt;T&gt;</code>	关系
<code>greater&lt;T&gt;</code>	关系
<code>greater_equal&lt;T&gt;</code>	关系
<code>less&lt;T&gt;</code>	关系
<code>less_equal&lt;T&gt;</code>	关系
<code>logical_and&lt;T&gt;</code>	逻辑
<code>logical_not&lt;T&gt;</code>	逻辑
<code>logical_or&lt;T&gt;</code>	逻辑
<code>minus&lt;T&gt;</code>	算术
<code>modulus&lt;T&gt;</code>	算术
<code>negate&lt;T&gt;</code>	算术
<code>not_equal_to&lt;T&gt;</code>	关系
<code>plus&lt;T&gt;</code>	算术
<code>multiplies&lt;T&gt;</code>	算术

图 20.41 标准库中的函数对象

```

1 // Fig. 20.42: fig20_42.cpp
2 // Demonstrating function objects.
3 #include <iostream>
4 #include <vector>
5 #include <algorithm>
6 #include <numeric>
7 #include <functional>
8
9 using namespace std;
10
11 // binary function adds the square of its second argument and
12 // the running total in its first argument and
13 // returns the sum
14 int sumSquares( int total, int value )
15 { return total + value * value; }
16
17 // binary function class template which defines an overloaded
18 // operator() that function adds the square of its second
19 // argument and the running total in its first argument and
20 // returns the sum
21 template< class T >
22 class SumSquaresClass : public binary_function< T, T, T >
23 {
24 public:
25     const T &operator()( const T &total, const T &value )
26     { return total + value * value; }
27 };
28
29 int main()
30 {
31     const int SIZE = 10;
32     int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
33     vector< int > v( a1, a1 + SIZE );
34     ostream_iterator< int > output( cout, " " );
35     int result = 0;

```

```

36
37  cout << "vector v contains:\n";
38  copy( v.begin(), v.end(), output );
39  result = accumulate( v.begin(), v.end(), 0, sumSquares );
40  cout << "\n\nSum of squares of elements in vector v using "
41        << "binary\nfunction sumSquares: " << result;
42
43  result = accumulate( v.begin(), v.end(), 0,
44                      SumSquaresClass< int >() );
45  cout << "\n\nSum of squares of elements in vector v using "
46        << "binary\nfunction object of type "
47        << "SumSquaresClass< int >: " << result << endl;
48  return 0;
49 }
vector v contains:
1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in vector v using binary
function sumSquares: 385

sum of squares of elements in vector v usnig binary
function objcet of type SumSquaresClass< int >: 385

```

图 20.42 演示二元函数对象

第 14 行和第 15 行:

```

int sumSquares( int total, int value )
{ return total + value * value; }

```

定义函数 `sumSquares` 计算第二个参数 `value` 的平方，将平方值与第一个参数 `total` 相加，并返回和。函数 `accumulate` 重复传递所迭代序列元素为第二个参数，本例中传给 `sumSquares`。首次调用 `sumSquares` 时，第一个参数是 `total` 的初始值（作为 `accumulate` 的第三个参数，本例中为 0）。后面再调用 `sumSquares` 时，第一个参数是前面调用 `sumSquares` 返回的动态和。`accumulate` 完成时，其返回序列中所有元素的平方和。

第 21 行到 27 行:

```

template< class T >
class SumSquaresClass : binary_function< T, T, T >
{
public:
    const T &operator()( const T &total, const T &value )
        { return total + value * value; }
};

```

定义一个 `SumSquaresClass` 类，从头文件 `<functional>` 中定义的 `binary_function` 类继承而来。从 `binary_function` 继承的类对重载的 `operator()` 函数（带有两个参数）进行定义。`SumSquaresClass` 类定义的函数对象让重载的 `operator()` 函数完成与函数 `sumSquares` 相同的任务。`binary_function` 模板的三个类型参数 (`T`) 是 `operator()` 的第一个参数、`operator()` 的第二个参数的类型和 `operator()` 的返回值类型。函数 `accumulate` 将所迭代序列元素作为第二个参数，并重复传递给对传入 `accumulate` 算法的

SumSquaresClass 类对象调用的 operator() 函数。首次调用 operator() 时, 第一个参数是 total 的初始值 (作为 accumulate 的第三个参数, 本例中为 0)。后面再调用 operator() 时, 第一个参数是前面调用 operator() 返回的动态和。accumulate 完成时, 其返回序列中所有元素的平方和。

第 39 行:

```
result = accumulate( v.begin(), v.end(), 0, sumSquares );
```

调用函数 accumulate, 用函数 sumSquares 的指针作为最后一个参数。

第 43 行和第 44 行:

```
result = accumulate( v.begin(), v.end(), 0,  
                    SumSquaresClass< int >() );
```

调用函数 accumulate, 用 SumSquaresClass 类对象作为最后一个参数。表达式 SumSquaresClass< int >() 生成 SumSquaresClass 类的实例并传入 accumulate, 用该对象调用 operator()。上述语句可以改写成下列两条语句:

```
SumSquaresClass< int > sumSquaresObj;  
result = accumulate( v.begin(), v.end(), 0, sumSquaresObj);
```

第一行定义 SumSquaresClass 类对象。第二行将该对象传入函数 accumulate, 并根据其调用重载的 operator() 函数。

软件工程视点 20.10

与函数指针不同的是, 函数对象还可以封装数据。

## 小结

- 利用 STL 可以节省大量时间和精力, 并得到更高质量的程序。
- 对任何特定应用程序, 可能适用几种不同的 STL 容器。选择最适合的容器, 使应用程序取得最佳性能。
- STL 容器都是“模板化”的, 可以调整成保存与特定应用程序相关的数据类型。
- STL 包括许多常用数据结构容器, 并提供很多算法, 使得程序处理这些容器中的数据。
- 容器分为三大类: 顺序容器、关联容器和容器适配器, 顺序容器和关联容器统称为第一类容器。
- 还有另外四种容器称为近容器, 因为它们与第一类容器提供类似功能, 但没有提供第一类容器的全部功能, 即数组、string、bitset 和 valarray。
- 可以在 vector 后面快速插入与删除, 而且能直接访问任何元素。vector 支持随机访问迭代器。
- 可以在 deque 前面或后面快速插入与删除, 而且能直接访问任何元素。deque 支持随机访问迭代器。
- 可以从 list 的任何地方快速插入与删除。list 支持双向迭代器。
- set 提供关键字的快速查找, 不允许重复值。set 支持双向迭代器。
- multiset 提供关键字的快速查找, 允许重复值。multiset 支持双向迭代器。
- map 提供关键字和其相应映射值的快速查找, 不允许重复值 (即一对一映射)。map 支持双向迭代器。

- `multimap` 提供关键字和其相应映射值的快速查找, 允许重复值 (即一对多映射)。`multimap` 支持双向迭代器。
- `stack` 提供后进先出的数据结构。
- `queue` 提供先进先出的数据结构。
- `priority_queue` 提供先进先出的数据结构, 高优先级项目总是放在 `priority_queue` 的前面。
- STL 经过认真设计, 使容器提供类似的功能。有许多一般化操作是所有容器都适用的, 而有些操作则适用于类似容器的子集。这样就可以用新类扩展 STL。
- STL 避免使用虚函数, 而用常规编程 (通过模板) 以达到更好的执行性能。
- 一定要保证容器中存放的元素类型支持最基本的功能集合, 包括复制构造函数和赋值运算符, 对于关联容器还应提供小于运算符 (<)。
- 迭代器用于序列, 这些序列可能在容器中, 也可能是输入序列或输出序列。
- 输入迭代器只能一次一个元素地向前移动 (即从容器开头到容器末尾)。输入迭代器只支持一遍算法。
- 输出迭代器只能一次一个元素地向前移动 (即从容器开头到容器末尾)。输出迭代器只支持一遍算法。
- 正向迭代器组合输入迭代器与输出迭代器的功能, 正向迭代器支持多遍算法。
- 双向迭代器组合正向迭代器的功能与逆向移动的功能。
- 随机访问迭代器组合双向迭代器的功能与直接访问容器中任何元素的功能, 即可以向前或向后跳过任意个元素。
- 数组指针可以代替所有 STL 算法中的迭代器。
- STL 包括大约 70 种标准算法。变化序列算法会修改容器内容, 而非变化序列算法不会修改容器内容。
- 函数 `fill` 和 `fill_n` 将容器中一定范围内的元素设置为特定值。
- 函数 `generate` 和 `generate_n` 用产生器函数对容器中一定范围内的元素生成各自的值。
- 用函数 `equal` 比较两个数值序列的相等性。
- 函数 `mismatch` 比较两个数值序列, 返回一个 `pair` 迭代器对, 表示每个序列中不匹配元素的位置。如果所有元素匹配, `pair` 包含对每个序列执行 `end` 函数的结果。
- 用函数 `lexicographical_compare` 比较两个字符数组的内容, 确定一个序列是否小于另一个序列 (类似于字符串比较)。
- 函数 `remove` 和 `remove_copy` 删除序列中符合指定值的所有元素。函数 `remove_if` 和 `remove_copy_if` 根据传入这些函数的一元判定函数的返回结果为 `true`, 将序列中对应的元素删除。
- 函数 `replace` 和 `replace_copy` 将序列中符合指定值的所有元素换成新值。函数 `replace_if` 和 `replace_copy_if` 根据传入这些函数的一元判定函数的返回结果为 `true`, 将序列中对应的元素替换为新值。
- 函数 `random_shuffle` 随机排序序列中的元素。
- 函数 `count` 计算序列中具有指定数值的元素个数。函数 `count_if` 根据传入该函数的一元判定函数的返回结果为 `true`, 计算序列中对应的元素个数。
- 函数 `min_element` 寻找序列中的最小值。函数 `max_element` 寻找序列中的最大值。

- 函数 `accumulate` 计算序列中的数值之和, 这个函数的第二个版本接收一个常用函数的指针, 该函数取两个参数并返回一个值, 确定序列中的元素如何累加。
- 函数 `for_each` 对序列中的每个元素采用一个常用函数。这个常用函数取一个参数(不能修改)并返回 `void`。
- 函数 `transform` 对序列中的每个元素采用一个常用函数。这个常用函数取一个参数(可以修改)并返回转换的值。
- 函数 `find` 寻找序列中的一个元素, 如果找到, 则返回该元素的迭代器, 否则 `find` 返回序列末尾的迭代器。函数 `find_if` 寻找所提供一元判定函数返回 `true` 的第一个元素。
- 函数 `sort` 按排序顺序排列序列元素(默认为升序, 也可以用二元判定函数表示顺序)。
- 函数 `binary_search` 确定元素是否在已排序的序列中。
- 函数 `swap` 交换两个值。
- 函数 `iter_swap` 交换迭代器所指的两个值。
- 函数 `swap_search` 交换一个序列中的元素与另一个序列中的元素。
- 函数 `copy_backward` 将一个序列中的元素放到另一个序列中, 从第二个序列的最后一个元素开始, 由后向前复制。
- 函数 `merge` 将两个升序排列序列合并为第三个升序排列序列。注意 `merge` 也适用于未排序序列, 但这时不产生排序序列。
- `back_inserter` 用容器的默认功能在容器末尾插入元素。当容器中没有更多元素时, 插入元素导致容器长度自动加大。还有另外两种插入器: `front_inserter` 和 `inserter`。`front_inserter` 在容器开头插入一个元素(用参数指定), `inserter` 在第一个参数提供的容器中第二个参数指定的迭代器之前插入元素。
- 函数 `unique` 删除排序序列中的所有重复值。
- 函数 `reverse` 逆转序列中的所有元素。
- 函数 `inplace_merge` 将两个排序的元素序列合并到同一个容器中。
- 函数 `unique_copy` 复制排序序列中的所有惟一元素。
- 函数 `reverse_copy` 逆向复制序列中的元素。
- 函数 `includes` 比较两个排序的数值集合, 确定一个集合是否包含另一集合。如果是, 则 `includes` 返回 `true`, 否则 `includes` 返回 `false`。
- 函数 `set_difference` 确定第一个排序值集合中不在第二个排序值集合中的元素(两个集合都应按升序排列)。
- 函数 `set_intersection` 确定第一个排序元素集合与第二个排序元素集合的交集(两个集合都应按升序排列)。
- 函数 `set_symmetric_difference` 确定第一个集合中不在第二个集合中的元素和第二个集合中不在第一个集合中的元素(两个集合都应按升序排列)。
- 函数 `set_union` 生成包含第一个集合和第二个集合所有元素的新集合(两个集合都应按升序排列)。
- 函数 `lower_bound` 确定排序数值序列中插入第三个参数而保持升序的第一个位置。
- 函数 `upper_bound` 确定排序数值序列中插入第三个参数而保持升序的最后一个位置。
- 函数 `equal_range` 返回一个正向迭代器对, 包含 `lower_bound` 和 `upper_bound` 操作的组合结果。

- 堆排序算法将元素数组排成特殊的二叉树,称为堆。堆的关键特性是最大元素总是在堆的顶上,二叉树中任何节点的子节点值总是小于或等于该节点的值。这种堆布置通常称为最大堆。
- 函数 `make_heap` 取一个数值序列并创建可以用来生成排序序列的堆。
- 函数 `sort_heap` 将已经形成堆的数值序列排序。
- 函数 `push_heap` 在堆中增加新值。`push_heap` 假设容器中的最后一个元素是刚刚加进堆中的元素,其他所有容器元素已经形成堆。
- 函数 `pop_heap` 删除堆顶上的元素。这个函数假设元素已经形成堆。
- 函数 `min` 确定两个值的最小值。
- 函数 `max` 确定两个值的最大值。
- `bitset` 类使位集合更易生成和操作。位集合用于表示位标志的集合。`bitset` 在编译时长度固定。

## 术语

<code>accumulate()</code>	<code>equal()</code>
adapter 适配器	<code>equal_range()</code>
<code>adjacent_difference()</code>	<code>erase()</code>
<code>adjacent_find()</code>	<code>fill()</code>
algorithm 算法	<code>fill_n()</code>
<code>&lt;algorithm&gt;</code>	<code>find()</code>
<code>assign()</code>	first-class containers 第一类容器
assignment 赋值	first-in-first-out (FIFO) 先进先出
associative array 关联数组	<code>for_each()</code>
associative container 关联容器	forward iterator 正向迭代器
<code>back()</code>	<code>front()</code>
<code>begin()</code>	function object 函数对象
bidirectional iterator 双向迭代器	<code>&lt;functional&gt;</code>
<code>binary_search()</code>	<code>generate()</code>
<code>const_iterator</code>	<code>generate_n()</code>
<code>const_reverse_iterator</code>	generic programming 常规编程
container 容器	<code>inplace_merge()</code>
container adapter classes 容器适配器类	input iterator 输入迭代器
<code>copy()</code>	<code>insert()</code>
<code>copy_backward()</code>	<code>istream_iterator</code>
<code>count()</code>	iterator 迭代器
<code>count_if()</code>	<code>&lt;iterator&gt;</code>
creating an association 生成关联	<code>iter_swap()</code>
<code>&lt;deque&gt;</code>	last-in-first-out (LIFO) 后进先出
<code>deque&lt;T&gt;</code>	<code>lexicographical_compare()</code>
<code>deque&lt;T&gt;::iterator</code>	<code>&lt;list&gt;</code>
deque sequence container deque 顺序容器	list sequence container list 顺序容器
<code>empty()</code>	<code>lower_bound()</code>
<code>end()</code>	<code>make_heap()</code>

<map>	push_back()
map associative container map 关联容器	push_front
max()	push_heap()
max_element()	queue container adapter class queue 容器适配器类
max_size()	random-access iterator 随机访问迭代器
merge()	random_shuffle()
min()	range 范围
min_element()	rbegin()
mismatch()	remove()
multimap associative container multimap 关联容器	remove_copy()
multiset associative container multiset 关联容器	remove_copy_if()
mutating-sequence algorithms 变化序列算法	remove_if()
namespace std	rend()
non-mutating-sequence algorithm 非变化序列算法	replace()
nth_element	replace()
<numeric>	replace_copy()
one-to-one mapping 一对一映射	replace_copy_if()
operator==( )	replace_if()
operator>( )	reverse()
operator>=( )	reverse_copy()
operator<( )	reverse iterator 逆向迭代器
operator<=( )	reverse_iterator
operator!=	reverse the contents of a container 逆转容器内容
ostream_iterator	rotate()
output iterator 输出迭代器	rotate_copy()
partial_sort()	sequence 顺序
partial_sort_copy()	sequence container 顺序容器
partial_sum()	sequential container 顺序性容器
partition()	<set>
platform-independent class libraies 平台无关类库	set associative container set 关联容器
platform-specific class libraries 平台特定类库	set_difference()
pop()	set_intersection()
pop_back()	set_symmetric_difference()
pop_fornt()	set_union()
pop_heap()	size()
priority_queue container adapter class	size_type
priority_queue 容器适配器类	sort()
push()	sort_heap()
	sorting algorithm 排序算法



<stack>	top()
stack container adapter class	stack 容器适配器类
transform()	transform()
standard template library (STL)	标准模板库
unique()	unique()
string	upper_bound()
string()	val_array
struct less<T>	value_type
swap()	<vector>
swap_range()	vector sequence container
	vector 顺序容器

## 自测练习

- 20.1 (判断对错) STL 大量使用继承和虚函数。
- 20.2 两种 STL 容器是顺序容器和 \_\_\_\_\_ 容器。
- 20.3 STL 不用 new 和 delete 而用 \_\_\_\_\_ 实现各种控制内存分配与释放的方法。
- 20.4 五种主要迭代器类型是 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- 20.5 (判断对错) 指针是迭代器的一般化形式。
- 20.6 (判断对错) STL 算法可以处理 C 语言式基于指针的数组。
- 20.7 (判断对错) STL 算法封装成每个容器类中的成员函数。
- 20.8 (判断对错) remove 算法不减小删除元素的 vector 长度。
- 20.9 STL 中用 \_\_\_\_\_ 对象进行内存分配与释放。
- 20.10 三种 STL 容器适配器是 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- 20.11 (判断对错) 容器成员函数 end() 得到容器中最后一个元素的位置。
- 20.12 STL 算法用 \_\_\_\_\_ 间接操作容器元素。
- 20.13 sort 算法要求 \_\_\_\_\_ 迭代器。

## 自测练习答案

- 20.1 不正确。这种做法对性能不利，因而不用。
- 20.2 关联。
- 20.3 分配器。
- 20.4 输入、输出、正向、双向、随机访问。
- 20.5 不正确。正好相反。
- 20.6 正确。
- 20.7 不正确。STL 算法不是成员函数，而是通过迭代器间接操作容器元素。
- 20.8 正确。
- 20.9 分配器。
- 20.10 stack、queue、priority\_queue。
- 20.11 不正确。实际上得到容器中最后一个元素后面一位的位置。
- 20.12 迭代器。
- 20.13 随机访问。

## 练习

- 20.14 编写一个函数模板 `palindrome`，取 `const vector` 参数并根据 `vector` 是否正向逆向都一样而返回 `true` 或 `false` 值（例如，包含 1、2、3、2、1 的 `vector` 是回文，而包含 1、2、3、4 的 `vector` 则不是）。
- 20.15 修改图 20.40 的 Eratosthenes 筛选法程序，使用户输入非质数数据时，程序显示该值的质数因子。记住，质数只能被 1 和本身整除。每个非质数都有惟一的质数因子分解式。例如，数值 54 的因子有 2、3、3、3。这些值相乘即可得到 54。对数值 54，输出的质数因子为 2 和 3。
- 20.16 修改练习 20.15，使用户输入非质数数据时，程序显示该值的质数因子和各个因子出现次数。例如，54 的输出为：

The unique prime factorization of 54 is: 2 \* 3 \* 3 \* 3

## Internet 和 World Wide Web 中的 STL 资源

下面列出 Internet 和 World Wide Web 中的 STL 资源。这些站点包括教程、参考资料、常见问题 (FAQ)、文章、书籍、访谈、ANSI/ISO C++ 草案标准和软件。

### 教程

[http:// w6.infosys.tuwien.ac.at/Research/Component/STL.newbie.html](http://w6.infosys.tuwien.ac.at/Research/Component/STL.newbie.html)

"Mumit's STL Newbie Guide" 是 STL 信息的宝贵资源。其中介绍的课题包括：编写容器对象、类的构造函数、类的运算符、指针与 STL、STL 容器中存放的指针包装、STL 容器中存放的派生对象、检查映射中的项目、`char*`、判定、比较器、常用函数、STL 迭代器、迭代器标志、链表间的复制、映射间的复制、STL 适配器、删除与擦除、排序和洗牌。

<http://www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html>

这个 STL 教程按例子、思路、组件和扩展 STL 进行组织。其中有使用 STL 组件的代码例子、重要解释和有用的示意图。

[http://web.fttech.net/~honeyg/articles/eff\\_stl.htm](http://web.fttech.net/~honeyg/articles/eff_stl.htm)

这个 STL 教程提供 STL 组件、容器、流与迭代器适配器、转换与选择数值、过滤与转换数值、对象等信息。

<http://www-leland.stanford.edu/~iburrell/cpp/stl.html>

这个站点提供 STL 资源的链接，包括 FAQ、FTP 站点和教程。

[http://web1.fttech.net/~honeyg/articles/tiny\\_stl.htm](http://web1.fttech.net/~honeyg/articles/tiny_stl.htm)

"A Tiny STL Primer" 简介容器成员、迭代器、算法与适配器代码例子，还有联机 STL 资源和书籍的简表。

[http://www.xraylith.wisc.edu/~khan/software/stl/os\\_examples/examples.html](http://www.xraylith.wisc.edu/~khan/software/stl/os_examples/examples.html)

这个站点对 STL 初学者很有帮助，有 STL 简介和 ObjectSpace STL Tool Kit 例子。

<http://www.cs.bham.ac.uk/~jdm/cpp.html>

这是 C++ 和 STL 信息最丰富的资源清单。STL 链接包括 FAQ、教程、文章、编译器等等。

#### 参考资料

[http://www.sgi.com/Technology/STL/other\\_resources.html](http://www.sgi.com/Technology/STL/other_resources.html)

这个站点列出 15 个 STL 相关的 Web 站点并推荐了一些 STL 书籍。

<ftp://ftp.bluenepetune.com/pub/users/yotam/stlqr-1.01.tar.gz>

<ftp://ftp.bluenepetune.com/pub/users/yotam/stlqr101.zip>

"STL-Quick-Reference version 1.01"

<http://www.cs.rpi.edu/projects/STL/stl/stl.html>

这是 Standard Template Library Online Reference 主页，详细介绍 STL 以及其他 STL 信息资源的链接。

<http://www.sgi.com/Technology/STL/>

Silicon Graphics Standard Template Library Programmer's Guide 是 STL 信息的重要资源，可以从这个站点下载 STL、寻找最新信息、设计文档和链接其他 STL 资源。

<http://www.ipmce.su/people/fbp/stl/stlport.html>

这个站点是精彩而新鲜的 STL 资源，其中的内容包括 STL 匹配信息、Testsuite for STLport、Exception-Handling Testsuite for STL、STL 文档、当前 STL 相关项目简介、已知缺陷的纠正方法、STL 支持的软件组件清单和其他 STL 相关站点清单。

#### 常见问题 (FAQ)

<http://www-leland.stanford.edu/~iburrell/cpp/stl.html>

这个站点提供 STL 资源的链接，包括 FAQ、FTP 站点和教程。

<http://www.cs.bham.ac.uk/~jdm/cpp.html>

这是 C++ 和 STL 信息最丰富的资源清单。STL 链接包括 FAQ、教程、文章、编译器等等。

<ftp://butler.hpl.hp.com/stl/stl.faq>

这是 Marian Corcoran 维护的 STL 常见问题及答案的 FTP 站点，Marian Corcoran 是 ANSI 会员和 C++ 专家。

#### 文章、书籍、访谈

[http://www.sgi.com/Technology/STL/other\\_resources.html](http://www.sgi.com/Technology/STL/other_resources.html)

这个站点列出 15 个 STL 相关 Web 站点并推荐了一些 STL 书籍。

<http://www.byte.com/art/9510/sec12/art3.htm>

Byte Magazine 站点提供了 Alexander Stepanov 所写的 STL 文章的副本。Alexander Stepanov 是标准模板库的创始人之一，他提供了常规编程中使用 STL 的信息。

<ftp://ftp.cs.rpi.edu/pub/stl/>

Hewlett-Packard 的 STL 文档和研究论文。这个站点还包括《STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library》(作者 D.R. Musser 和 Atul Saini, Addison-Wesley, Reading, MA, 1996) 的源代码。

<http://www.ipmc.su/people/fbp/stl/StepanovUSA.html>  
<http://www.sgi.com/Technology/STL/drdoobbs-interview.html>

这些与 Alexander Stepanov 的访谈介绍了创建 Standard Template Library 的一些有趣信息。Alexander Stepanov 介绍了 STL 如何概念化、常规编程、缩略语 "STL" 的由来等等。

<http://www.cs.bham.ac.uk/~jdm/cpp.html>

这是 C++ 和 STL 信息最丰富的资源清单。STL 链接包括 FAQ、教程、文章、编译器等等。

### ANSI/ISO C++ Draft Standard (草案标准)

<http://www.maths.warwick.ac.uk/cpp/pub/wp/html/cd2/index.html>

这个站点有 "Draft Proposed International Standard for Information Systems-Programming Language C++" 的工作报告。

<http://math.nist.gov/tnt/stl.html>

这个站点是 C++ Standard Template Library 的信息资源, 链接 HP STL 的 Web 站点、Standard Template Library Online Reference 主页、ANSI C++ Draft Standard 和推荐书籍。

<http://ourworld.compuserve.com/homepages/bloem/bbstl.htm>

下载 Stepanov 和 lee 的 Standard Template Library 文档和 Proposed C++ Draft Standard 文档。

<http://www.dinkumware.com/refcpp.html>

这个站点包含 ANSI/ISO Draft Standard C++ Library 的信息和标准模板库的丰富信息。

<http://www.cygnum.com/misc/wp/dec96pub/>

这个站点有 "Draft Proposed International Standard for Information Systems-Programming Language C++" 工作报告的联机副本。

### 软件

<http://www.cs.rpi.edu/~musser/stl.html>

RPI STL 站点包括一些 STL 与其他 C++ 库的差别信息, 如何编译使用 STL 的程序, 列出 STL 包括的主要文件、使用 STL 的实例程序、STL 容器类和 STL 迭代器类别, 并提供兼容 STL 的编译器清单、STL 源代码和相关材料的 FTP 站点。

<ftp://ftp.cs.rpi.edu/pub/stl/>

Hewlett-Packard 的 STL 文档和研究论文。这个站点还包括《STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library》(作者 D.R. Musser 和 Atul Saini, Addison-Wesley, Reading, MA, 1996) 的源代码。

<http://www.cyberport.com/~tangent/programming/stl/resources.html>

这个 STL 资源站点包括兼容 STL 的编译器清单和几十个其他 STL 相关站点的链接。

<http://www.mathcs.sjsu.edu/faculty/horstman/safestl.html>

下载 SAFESTL.ZIP, 这是在使用 STL 的程序中寻找错误的工具。

<http://www.sirius.com/~ouchida/>

Silicon Graphics 公司的 Microsoft Visual C++ 5.0 使用的 STL。

<http://www.objectspace.com/jgl/>

ObjectSpace 提供将 C++ 移植到 Java 的信息, 可以免费下载 Standards<ToolKit>可移植类库。这个工具库的关键特性包括容器、迭代器、算法、分配器、字符串和异常。

<http://www.cs.bham.ac.uk/~jdm/cpp.html>

这是 C++ 和 STL 信息最丰富的资源清单。STL 链接包括 FAQ、教程、文章、编译器等等。

<http://www.cs.rpi.edu/~wised/stl-borland.html.html>

“Using the Standard Template Library with Borland C++”。这个站点是使用 Borland C++ 编译器的重要参考资料。作者对警告和不兼容性进行了讨论。

<http://www.geocities.com/SiliconValley/Pines/2010/note.txt>

汇集了 STL 与 Microsoft Developer's C++ 4.2 之间的不兼容性。

<http://www.microsoft.com/visualc/legacy/v4.0/pc/techinfo/stlchg.htm>

Microsoft 的 “MFC and Standard Template Library (STL) Alert” 站点介绍了 Visual C++ 用户使用 Visual C++ 中的 MFC 和 STL 时可能遇到的一些问题。Microsoft 公司还提供其他 C++ 资源的链接。

## STL 文献

- (Am97) Ammeraal, L., *STL for C++ Programmers*, New York, NY: John Wiley, 1997.
- (C++97) X3 Secretariat: *Draft Standard—The C++ Language*. X3J16/97-14882. Information Technology Council (NSITC), Washington, DC, USA: 1997.
- (G195) Glass, G. and B. Schuchert, *The STL<Primer>*, Upper Saddle River, NJ: Prentice Hall PTR, 1995.
- (He97) Henricson, M., and E. Nyquist, *Industrial Strength C++: Rules and Recommendations*, Upper Saddle River, NJ: Prentice Hall, 1997.
- (Ko97) Koenig, A. and B. Moo, *Ruminations on C++*, Reading, MA: Addison-Wesley, 1997.
- (Mu94) Musser, D.R. and A.A. Stepanov, “Algorithm-Oriented Generic Libraries”, *Software Practice and Experience*, Vol.24, No.7 July 1994.
- (Mu96) Musser, D.R. and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Reading, MA: Addison-Wesley Publishing Company, 1996.
- (Ne95) Nelson, M., *C++ Programmer's Guide to the Standard Template Library*, Foster City, CA: programmers press, a Division of IDG Books Worldwide, Inc., 1995.

- 
- ( Po97 ) Pohl, I., *C++Distilled: A Concise ANSI/ISO Reference and Style Guide*, Reading, MA: Addison-Wesley 1997.
  - ( Po97a ) Pohl, I., *Object-Oriented Programming Using C++*, Second Edition, Reading, MA: Addison-Wesley Publishing Company, 1997.
  - ( St95 ) Stepanov, A. and M.Lee, "The Standard Template Library", *Internet Distribution*, Published at <ftp://butler.hpl.hp.com/stl>, July 7, 1995.
  - ( Sr94 ) Stroustrup, B., "Making a vector Fit for a Standard", *The C++ Report*, October 1994.
  - ( Sr94a ) Stroustrup, B., *The Design Evolution of C++*, Reading, MA: Addison-Wesley Publishing Company, 1994.
  - ( Sr97 ) Stroustrup, B., *The C++ Programming Language*, Third Edition, Reading, MA: AddisonWesley Publishing Company, 1997.
  - ( Vi94 ) Vilot, M.J., "An Introduction to the Standard Template Library", *The C++ Report*, Vol.6, No.8, October 1994.

## 第 21 章 ANSI/ISO C++ 标准语言补充

### 教学目标

- 了解与使用 bool 数据类型
- 使用强制类型转换运算符 static\_cast、const\_cast 和 reinterpret\_cast
- 了解 namespace 的概念
- 了解和使用运行时类型信息 (RTTI) 与运算符 typeid 和 dynamic\_cast
- 了解运算符关键字
- 了解 explicit 构造函数
- 在 const 对象中使用 mutable 成员
- 了解与使用类成员指针运算符.\* 和 ->\*
- 了解 virtual 基类在多重继承中的作用

### 21.1 简介

下面要考虑一些新的 ANSI/ISO C++ 草案标准特性, 包括 bool 数据类型、强制类型转换运算符、名字空间、运行时类型信息 (RTTI) 和运算符关键字, 我们还要介绍类成员指针运算符和 virtual 基类 (这是 ANSI C++ 中的新特性)。本章是与 Deitel & Associates 公司的同事 Tem Nieto 合作编写的。

### 21.2 bool 数据类型

ANSI/ISO C++ 草案标准提供的 bool (布尔) 数据类型可取值为 false 或 true, 而旧式方法则用 0 表示 false, 用非 0 表示 true。图 21.1 的程序演示了 bool 数据类型。

```
1 // Fig. 21.1: fig21_01.cpp
2 // Demonstrating data type bool.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     bool boolean = false;
10    int x = 0;
11
12    cout << "boolean is " << boolean
13         << "\nEnter an integer: ";
14    cin >> x;
15
```

```

16     cout << "integer " << x << " is"
17         << ( x ? " nonzero " : " zero " )
18         << "and interpreted as ";
19
20     if ( x )
21         cout << "true\n";
22     else
23         cout << "false\n";
24
25     boolean = true;
26     cout << "boolean is " << boolean;
27     cout << "\nboolean output with boolalpha manipulator is "
28         << boolalpha << boolean << endl;
29
30     return 0;
31 }

```

**输出结果：**

```

boolean is 0
Enter an integer: 22
integer 22 is nonzero and interpreted as true
boolean is 1
boolean output with boolalpha manipulator is true

```

图 21.1 演示 bool 数据类型

**第 9 行：**

```
bool boolean = false;
```

声明变量 `boolean` 的类型为 `bool`，并将 `boolean` 初始化为 `false`。变量 `x` 声明并初始化为 0。第 12 行：

```
cout << "boolean is " << boolean
```

输出 `boolean` 值。输出数值 0 而不是关键字 `false`。`bool` 的默认显示为数字值。

第 20 行用 `x` 值（在第 14 行输入）作为 `if/else` 条件。如果 `x` 为 0，则条件为 `false`，否则条件为 `true`。注意负数为非 0 值，因此为 `true`。

第 25 行将 `true` 赋给 `boolean`。`boolean` 的值（1）在第 26 行输出。`bool` 变量默认输出为 0 和 1。重载流插入运算符 `<<` 将 `bool` 显示为整数。

**第 27 行和第 28 行：**

```
cout << "\nboolean output with boolalpha manipulator is "
    << boolalpha << boolean << endl;
```

用流操纵算子 `boolalpha` 将输出流设置成将 `bool` 值显示为字符串“`true`”和“`false`”。流操纵算子 `boolalpha` 也可以用于输入。

指针、`int`、`float` 等等可以隐式转换为 `bool`。0 值转换为 `false`，非 0 值转换为 `true`。例如，下列表达式：

```
bool dc = false + x * 2 - b && true;
```



在 `x` 为 3、`b` 为 `true` 时将 `dc` 赋值为 `true`。注意赋值表达式右边求值为 5，但这个值可以隐式转换为 `true`。

#### 编程技巧 21.1

生成表示真假值的静态变量时，最好用 `bool` 而不用 `int`。

#### 编程技巧 21.2

用 `true` 和 `false` 而不是用 0 和非 0 值能使程序更清晰。

## 21.3 `static_cast` 运算符

ANSI/ISO C++ 草案标准引入 4 个新的强制类型转换运算符，优于 C 和 C++ 中使用的旧式强制类型转换。新的强制类型转换比旧式的功能更弱也更具体，可以更好地控制程序。使用强制类型转换时必须小心，它常常是错误的来源，因此新式强制类型转换也更容易用自动工具寻找和确定。新式强制类型转换的另一优点是 4 个强制类型转换的用途完全分开，而旧式强制类型转换则是比较笼统的。

C++ 提供了在两种类型之间转换的 `static_cast` 运算符。类型检查在编译时进行。`static_cast` 运算符进行标准转换（例如 `void*` 换算为 `char*`、`int` 换算为 `float` 等等）及其逆转换。图 21.2 的程序演示 `static_cast` 运算符。

#### 常见编程错误 21.1

用 `static_cast` 运算符进行非法的类型转换是个语法错误。非法的类型转换包括将 `const` 类型转换为非 `const` 类型，转换不用 `public` 继承相关联的类型之间的指针和引用，转换为没有相应构造函数或转换运算符的类型。

```
1 // Fig. 21.2: fig21_02.cpp
2 // Demonstrating the static_cast operator.
3 #include <iostream.h>
4
5 class BaseClass {
6 public:
7     void f( void ) const { cout << "BASE\n"; }
8 };
9
10 class DerivedClass: public BaseClass {
11 public:
12     void f( void ) const { cout << "DERIVED\n"; }
13 };
14
15 void test( BaseClass * );
16
17 int main()
18 {
19     // use static_cast for a conversion
20     double d = 8.22;
21     int x = static_cast< int >( d );
22
23     cout << "d is " << d << "\nx is " << x << endl;
24
25     BaseClass base; // instantiate base object
26     test( &base ); // call test
```

```
27
28     return 0;
29 }
30
31 void test( BaseClass *basePtr )
32 {
33     DerivedClass *derivedPtr;
34
35     // cast base class pointer into derived class pointer
36     derivedPtr = static_cast< DerivedClass * >( basePtr );
37     derivedPtr->f();    // invoke DerivedClass function f
38 }
```

**输出结果:**

```
d is 8.22
x is 8
DERIVED
```

图 21.2 演示 static\_cast 运算符

程序声明 BaseClass 和 DerivedClass 类。每个类定义一个成员函数 f。第 20 行和第 21 行:

```
double d = 8.22;
int x = static_cast< int >( d );
```

声明并初始化 d 与 x。static\_cast 运算符将 d 由 double 换算为 int。该运算符可以用于 int、double、float 等基础数据类型之间的大多数转换。

#### 软件工程视点 21.1

ANSI/ISO C++ 草案标准增加新的强制类型转换运算符 (如 static\_cast 运算符) 之后, 旧式 C 语言的强制类型转换已经过时。

#### 编程技巧 21.3

最好用更安全更可靠的 static\_cast 运算符而不用旧式 C 语言强制类型转换运算符。

第 25 行将 BaseClass 对象 base 实例化, 并在第 26 行按引用传递给函数 test。在指针 basePtr 中接收传入 test 的地址。第 33 行声明 DerivedClass 的指针 derivedPtr。第 36 行:

```
derivedPtr = static_cast< DerivedClass * >( basePtr );
```

用 static\_cast 向下进行类型转换, 从 BaseClass \* 转换到 DerivedClass \*。第 9 章曾介绍过, 从基类指针向下转换到派生类指针是个危险的操作, 但 static\_cast 允许这个转换。函数 f 由 derivedPtr 调用 (第 37 行)。

## 21.4 const\_cast 运算符

C++ 提供 const\_cast 运算符, 用于转换 const 或 volatile。图 21.3 的程序演示 const\_cast 运算符。

```
1 // Fig. 21.3: fig21_03.cpp
2 // Demonstrating the const_cast operator.
```

```
3 #include <iostream.h>
4
5 class ConstCastTest {
6 public:
7     void setNumber( int );
8     int  getNumber() const;
9     void printNumber() const;
10 private:
11     int number;
12 };
13
14 void ConstCastTest::setNumber( int num ) { number = num; }
15
16 int ConstCastTest::getNumber() const { return number; }
17
18 void ConstCastTest::printNumber() const
19 {
20     cout << "\nNumber after modification: ";
21
22     // the expression number-- would generate compile error
23     // undo const-ness to allow modification
24     const_cast< ConstCastTest * >( this )->number--;
25
26     cout << number << endl;
27 }
28
29 int main()
30 {
31     ConstCastTest x;
32     x.setNumber( 8 ); // set private data number to 8
33
34     cout << "Initial value of number: " << x.getNumber();
35
36     x.printNumber();
37     return 0;
38 }
```

**输出结果:**

```
Initial value of number: 8
Number after modification: 7
```

图 21.3 演示 const\_cast 运算符

第 5 行到第 12 行声明的 ConstCastTest 类包含三个成员函数和 private 变量 number。两个成员函数声明为 const。函数 setNumber 设置 number 值，函数 getNumber 返回 number 值。

const 成员函数 printNumber 修改 number 值（第 24 行）：

```
const_cast< ConstCastTest * >( this )->number--;
```

在 const 成员函数 printNumber 中，this 指针的数据类型为 const 类型的 ConstCastTest \*。上述语句用 const\_cast 运算符强制转换 this 指针的常量性。这个语句余下部分的该指针类型为 ConstCastTest \*，允许修改 number。const\_cast 运算符不能直接转换常量变量的常量性。

## 21.5 reinterpret\_cast 运算符

C++ 提供 `reinterpret_cast` 运算符用于非标准强制类型转换（如 `void *` 转换为 `int` 等等），`reinterpret_cast` 运算符也可以用于标准强制类型转换（如 `double` 转换为 `int` 等等）。图 21.4 的程序演示 `reinterpret_cast` 运算符的使用。

```
1 // Fig. 21.4: fig21_04.cpp
2 // Demonstrating reinterpret_cast operator.
3 #include <iostream.h>
4
5 int main()
6 {
7     unsigned x = 22, *unsignedPtr;
8     void *voidPtr = &x;
9     char *charPtr = "C++";
10
11     // cast from void * to unsigned *
12     unsignedPtr = reinterpret_cast< unsigned * >( voidPtr );
13
14     cout << "*unsignedPtr is " << *unsignedPtr
15          << "\ncharPtr is " << charPtr;
16
17     // use reinterpret_cast to cast a char * pointer to unsigned
18     cout << "\nchar * to unsigned results in: "
19          << ( x = reinterpret_cast< unsigned >( charPtr ) );
20
21     // cast unsigned back to char *
22     cout << "\nunsigned to char * results in: "
23          << reinterpret_cast< char * >( x ) << endl;
24
25     return 0;
26 }
```

**输出结果：**

```
* unsignedPtr is 22
charPtr is C++
char * to unsigned results in: 4287842
unsigned to char * results in: C++
```

图 21.4 演示 `reinterpret_cast` 运算符

程序声明一个整数和三个指针。指针 `voidPtr` 初始化为 `unsigned x` 的地址，指针 `charPtr` 初始化为 C 语言式字符串 “C++”。第 12 行：

```
unsignedPtr = reinterpret_cast< unsigned * >( voidPtr );
```

用运算符 `reinterpret_cast` 将 `voidPtr`（`void *` 类型）转换为 `unsigned *` 类型的指针 `unsignedPtr`。

第 18 行和第 19 行：

```
cout << "\nchar * to unsigned results in: "
     << ( x = reinterpret_cast< unsigned >( charPtr ) );
```

将转换 `charPtr` 的结果输出到 `unsigned`。该结果赋给 `x`，表示字符串“C++”的十进制地址位置。

第 22 行和第 23 行：

```
cout << "\nunsigned to char * results in: "  
      << reinterpret_cast< char * >( x ) << endl;
```

将 `x` 值转换为 `char*`。转换结果是个地址 (`char*`)，产生 C 语言式字符串输出。

#### 测试与调试提示 21.1

使用运算符 `reinterpret_cast` 很容易进行危险的操作，造成严重的运行时错误。

#### 可移植性提示 21.1

运算符 `reinterpret_cast` 可能在不同平台上形成不同表现。

## 21.6 名字空间

一个程序在不同范围中包括许多不同标识符。有时一个范围中的变量会与另一范围中的变量重名，从而可能造成问题。这种重名可能在几个层次发生。标识符重名经常发生在第三方库使用重名的全局标识符（如函数）。发生这种情况时，通常会产生编译错误。

要解决这个问题，编译器厂家将全局标识符名用下划线（`_`）开头，第三方厂家的全局标识符用特定前缀字符开头。这种方法要求程序员不用下划线或第三方厂家使用的前缀字符作为标识符前缀。

#### 编程技巧 21.4

尽量不用下划线作为标识符的开头，以免造成连接器错误。

ANSI/ISO 草案标准的 C++ 试图用名字空间（`namespace`）解决这个问题。每个 `namespace` 定义一个全局标识符和全局变量范围。要使用 `namespace` 成员，就要用 `namespace` 名限定成员名，中间加上二元作用域运算符（`::`），如下所示：

```
namespace_name::member
```

或在使用这个名称之前用一个 `using` 语句。`using` 语句通常放在使用 `namespace_name` 成员的文件开头。例如，源代码文件开头用下列语句：

```
using namespace namespace_name;
```

指定 `namespace` 的成员 `namespace_name` 在文件中使用时无必加上 `namespace_name` 和二元作用域运算符（`::`）。

#### 编程技巧 21.5

尽量在成员前面加上 `namespace` 名和作用域运算符（`::`）以避免产生范围冲突。

并不是所有 `namespace` 都保证惟一性。两个第三方厂家可能使用相同的 `namespace`。图 21.5 演示了 `namespace` 的用法。

```
1 // Fig. 21.5: fig21_05.cpp  
2 // Demonstrating namespaces.  
3 #include <iostream>
```

```
4 using namespace std; // use std namespace
5
6 int myInt = 98;      // global variable
7
8 namespace Example {
9     const double PI = 3.14159;
10    const double E = 2.71828;
11    int myInt = 8;
12    void printValues();
13
14    namespace Inner { // nested namespace
15        enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
16    }
17 }
18
19 namespace { // unnamed namespace
20     double d = 88.22;
21 }
22
23 int main(
24 {
25     // output value d of unnamed namespace
26     cout << "d = " << d;
27
28     // output global variable
29     cout << "\n(global) myInt = " << myInt;
30
31     // output values of Example namespace
32     cout << "\nPI = " << Example::PI << "\nE = "
33         << Example::E << "\nmyInt = "
34         << Example::myInt << "\nFISCAL3 = "
35         << Example::Inner::FISCAL3 << endl;
36
37     Example::printValues(); // invoke printValues function
38
39     return 0;
40 }
41
42 void Example::printValues()
43 {
44     cout << "\n\nIn printValues:\n" << "myInt = "
45         << myInt << "\nPI = " << PI << "\nE = "
46         << E << "\nd = " << d << "\n(global) myInt = "
47         << ::myInt << "\nFISCAL3 = "
48         << Inner::FISCAL3 << endl;
49 }
```

**输出结果:**

```
d = 88.22
(global) myInt = 98
PI = 3.14159
E = 2.71828
myInt = 8
FISCAL3 = 1992
```

```
In printValues:
myInt = 8
PI = 3.14159
E = 2.71828
d = 88.22
(global) myInt = 98
FISCAL3 = 1992
```

图 21.5 演示 namespace

第4行:

```
using namespace std;
```

告诉编译器要用 namespace std。头文件<iostream>的内容都在 namespace std 中定义。

using namespace 语句指定程序中经常使用 namespace 的成员。这样,程序员可以访问 namespace 的所有成员,写成更简练的:

```
cout << "d =" << d;
```

而不必用:

```
std::cout << "d =" << d;
```

如果没有第4行,则图21.5中每个 cout 和 endl 都要限定 std:: 字样。C++ 的许多程序员还是习惯写成 std::cout。using namespace 语句可以用于预定义的 namespace(如 std) 或程序员定义的 namespace。

第8行到第17行:

```
namespace Example {
    const double PI = 3.14159;
    const double E = 2.71828;
    int myInt = 8;
    void printValues();

    namespace Inner { // nested namespace
        enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
    }
}
```

用关键字 namespace 定义 namespace Example。namespace 的程序体放在花括号中({})。与类体不同的是,namespace 程序体不用分号结尾。Example 的成员包括两个常量(PI 和 E)、一个 int 类型的值(myInt)、一个函数(printValues) 和一个嵌套 namespace (Inner)。注意成员 myInt 与全局变量 myInt 同名。同名变量应使用不同范围,否则是个语法错误。namespace 可以包含常量、数据、类、嵌套 namespace、函数等等。namespace 的定义应占有全局范围或其他 namespace 中的嵌套范围。

第19行到第21行:

```
namespace {
    double d = 88.22;
}
```

生成包括成员 d 的无名 namespace。无名 namespace 成员占有全局 namespace,可以直接访问而不必用 namespace 名限定。全局变量也是全局 namespace 的一部分,可以在文件声明之后的所有范围中访问。

**软件工程视点 21.2**

每个单独编译单元有自己的惟一无名 namespace，即无名 namespace 取代 static 连接说明符。

第26行输出d值。成员d可以作为无名 namespace 的成员直接访问。第29行输出全局变量 myInt 的值。第32行到第35行：

```
cout << "\nPI = " << Example::PI << "\nE = "
    << Example::E << "\nmyInt = "
    << Example::myInt << "\nFISCAL3 = "
    << Example::Inner::FISCAL3 << endl;
```

输出 PI、E、myInt 和 FISCAL3 的值。PI、E 和 myInt 是 Example 成员，因此用 Example:: 限定。因为有同名全局变量，因此成员 myInt 要限定，否则输出全局变量的值。FISCAL3 是嵌套 namespace Inner 的成员，用 Example::Inner:: 限定。

函数 printValues 是 Example 的成员，可以不用 namespace 限定符直接访问同一 namespace 中的其他成员，第44行的 cout 输出 myInt、PI、E、d 和全局变量 myInt 与 FISCAL3。注意，PI 和 E 不用 Example 限定，d 仍可访问，全局 myInt 用一元作用域运算符 (::) 限定，FISCAL3 用 Inner:: 限定。访问嵌套 namespace 的成员时，该成员要用 namespace 名限定（除非在嵌套 namespace 内）。

也可以通过 using 关键字使用各个 namespace 成员。例如，下列语句：

```
using Example::PI;
```

可以不用 namespace 限定而使用 PI。这通常在只有一个经常使用的 namespace 时进行。名字空间可以指定别名。例如，下列语句：

```
namespace CPPHTP2 = CplusplusHowToProgram2;
```

生成 CplusplusHowToProgram2 的别名 CPPHTP2。

**常见编程错误 21.2**

将 main 放在 namespace 中是个语法错误。

**软件工程视点 21.3**

大程序中最好每个实体在一个类、函数、块或 namespace 中声明，使每个实体的作用一目了然。

## 21.7 运行时类型信息 (RTTI)

运行时类型信息 (RTTI) 提供了运行时确定对象类型的方法。本节介绍两个重要的 RTTI 运算符 typeid 和 dynamic\_cast。图 21.6 演示 typeid。图 21.7 演示 dynamic\_cast。

**测试与调试提示 21.2**

为了使用 RTTI，有些编译器中需要启用 RTTI 功能。详见编译器的 RTTI 使用文档。

```
1 // Fig. 21.6: fig21_06.cpp
2 // Demonstrating RTTI capability typeid.
3 #include <iostream.h>
4 #include <typeinfo.h>
5
```



```
6 template < typename T >
7 T maximum( T value1, T value2, T value3 )
8 {
9     T max = value1;
10
11     if ( value2 > max )
12         max = value2;
13
14     if ( value3 > max )
15         max = value3;
16
17     // get the name of the type (i.e., int or double)
18     const char *dataType = typeid( T ).name();
19
20     cout << dataType << "s were compared.\nLargest "
21         << dataType << " is ";
22
23     return max;
24 }
25
26 int main()
27 {
28     int a = 8, b = 88, c = 22;
29     double d = 95.96, e = 78.59, f = 83.89;
30
31     cout << maximum( a, b, c ) << "\n";
32     cout << maximum( d, e, f ) << endl;
33
34     return 0;
35 }
```

**输出结果：**

```
ints were compared.
Largest int is 88
doubles were compared.
Largest double is 95.96
```

图 21.6 演示 typeid

第 4 行包括<typeinfo.h>头文件 (ANSI/ISO C++ 草案标准中为<typeinfo>), 其定义 typeid。程序定义一个函数模板 maximum, 取三个指定数据类型 T 的参数, 确定和返回其最大值。typename 关键字代替 class 关键字。这里, typename 与 class 的作用相同。

第 18 行:

```
const char *dataType = typeid( T ).name();
```

用函数 name 返回实现方法定义的 C 语言式字符串, 表示 T 的数据类型。编译时运算符 typeid 返回 type\_info 对象的引用。type\_info 对象是个系统维护的对象, 表示一种类型。注意, name 返回的字符串由系统拥有, 程序员不能将其删除。

## 编程技巧 21.6

在 switch 之类的测试中使用 RTTI 是对运行时类型信息 (RTTI) 的误用, 应使用虚函数。

运算符 `dynamic_cast` 用于多态编程中保证在运行时发生正确的转换 (即编译器无法验证是否发生正确的转换)。运算符 `dynamic_cast` 常用于从多态编程基类指针向派生类指针的向下类型转换。图 21.7 的程序演示运算符 `dynamic_cast`。

```
1 // Fig. 21.7: fig21_07.cpp
2 // Demonstrating dynamic_cast.
3 #include <iostream.h>
4
5 const double PI = 3.14159;
6
7 class Shape {
8     public:
9         virtual double area() const { return 0.0; }
10 };
11
12 class Circle: public Shape {
13     public:
14         Circle( int r = 1 ) { radius = r; }
15
16         virtual double area() const
17         {
18             return PI * radius * radius;
19         };
20     protected:
21         int radius;
22 };
23
24 class Cylinder: public Circle {
25     public:
26         Cylinder( int h = 1 ) { height = h; }
27
28         virtual double area() const
29         {
30             return 2 * PI * radius * height +
31                 2 * Circle::area();
32         }
33     private:
34         int height;
35 };
36
37 void outputShapeArea( const Shape * );    // prototype
38
39 int main()
40 {
41     Circle circle;
42     Cylinder cylinder;
43     Shape *ptr = 0;
44
45     outputShapeArea( &circle );    // output circle's area
46     outputShapeArea( &cylinder );  // output cylinder's area
```

```

47     outputShapeArea( ptr );           // attempt to output area
48     return 0;
49 }
50
51 void outputShapeArea( const Shape *shapePtr )
52 {
53     const Circle *circlePtr;
54     const Cylinder *cylinderPtr;
55
56     // cast Shape * to a Cylinder *
57     cylinderPtr = dynamic_cast< const Cylinder * >( shapePtr );
58
59     if ( cylinderPtr != 0 ) // if true, invoke area()
60         cout << "Cylinder's area: " << cylinderPtr->area();
61     else { // shapePtr does not refer to a cylinder
62
63         // cast shapePtr to a Circle *
64         circlePtr = dynamic_cast< const Circle * >( shapePtr );
65
66         if ( circlePtr != 0 ) // if true, invoke area()
67             cout << "Circle's area: " << circlePtr->area();
68         else
69             cout << "Neither a Circle nor a Cylinder.";
70     }
71
72     cout << endl;
73 }

```

**输出结果:**

```

Circle's area: 3.14159
Cylinder's area: 12.5664
Neither a Circle nor a Cylinder.

```

图 21.7 演示 dynamic\_cast

程序在第 7 行定义基类 Shape，包含虚函数 area、第 12 行继承 Shape 的派生类 Circle 和第 24 行继承 Circle 的派生类 Cylinder（均为 public 继承）。Circle 和 Cylinder 重定义函数 area。

在第 41 行到 43 行的函数 main 中，实例化 Circle 类的对象 circle 和 Cylinder 类的对象 cylinder，并将 Shape 类的指针 ptr 声明和初始化为 0。第 45 行到 47 行调用函数 outputShapeArea（第 51 行定义）三次。每次调用 outputShapeArea 时都显示三个结果之一：Circle 的面积、Cylinder 的面积或表示 Shape 不是 Circle 或 Cylinder 的指示信息。函数 outputShapeArea 接收 Shape 的指针参数，第一次调用接收 circle 的地址，第二次调用接收 cylinder 的地址，第三次调用接收 Shape 基类的指针 ptr。

第 57 行:

```
cylinderPtr = dynamic_cast< const Cylinder * >( shapePtr );
```

动态将 shapePtr（一种 const Shape\* 类型）向下转换为 const Cylinder\* 类型，这是用强制类型转换运算符 dynamic\_cast 进行的。结果，将 cylinder 对象的地址赋给 cylinderPtr，如果 Shape 不是 Cylinder 则 cylinderPtr 为 0。如果转换结果不是 0，则输出 Cylinder 的面积。

第 64 行:

```
circlePtr = dynamic_cast< const Circle * >( shapePtr );
```

动态将 shapePtr 向下转换为 const Circle \* 类型，这是用强制类型转换运算符 dynamic\_cast 进行的。结果，将 circle 对象的地址赋给 circlePtr，如果 Shape 不是 Circle 则 circlePtr 为 0。如果转换结果不是 0，则输出 Circle 的面积。

#### 常见编程错误 21.3

想对 void\* 类型的指针使用 RTTI 是个语法错误。

#### 软件工程视点 21.4

RTTI 常用于多态继承层次（通过虚函数）。

## 21.8 运算符关键字

ANSI/ISO C++ 草案标准提供了运算符关键字（如图 21.8），可以代替几个 C++ 运算符。运算符关键字可以在不支持!、&、^、~、|，等字符的键盘上编程时使用。

图 21.9 的程序演示了运算符关键字的用法。这个程序用 Microsoft Visual C++ 5.0 编译，要求包含头文件<iso646.h>才能使用运算符关键字。其他编译器中的头文件可能不同，因此，要检查编译器文档，确定所包含的头文件（某些编译器可能不需要任何头文件即可使用这些关键字）。

运算符	运算符关键字	说明
逻辑运算符关键字		
&&	and	逻辑 AND
	or	逻辑 OR
!	not	逻辑 NOT
不等性运算符关键字		
!=	not_eq	不等性
位运算符关键字		
&	bitand	位 AND（位与）
	bitor	位 OR（位或）
^	xor	位异 OR（位异或）
~	compl	位取反
位赋值运算符关键字		
&=	and_eq	位 AND 赋值
=	or_eq	位 OR 赋值
^=	xor_eq	位异 OR 赋值

图 21.8 用运算符关键字代替运算符符号

```
1 // Fig. 21.9: fig21_09.cpp
2 // Demonstrating operator keywords.
3 #include <iostream>
4 #include <iomanip>
5 #include <iso646.h>
6 using namespace std;
7
8 int main()
9 {
```

```

10  int a = 8, b = 22;
11
12  cout << boolalpha
13      << "    a and b: " << ( a and b )
14      << "\n    a or b: " << ( a or b )
15      << "\n    not a: " << ( not a )
16      << "\na not_eq b: " << ( a not_eq b )
17      << "\na bitand b: " << ( a bitand b )
18      << "\na bit_or b: " << ( a bitor b )
19      << "\n    a xor b: " << ( a xor b )
20      << "\n    compl a: " << ( compl a )
21      << "\na and_eq b: " << ( a and_eq b )
22      << "\n a or_eq b: " << ( a or_eq b )
23      << "\na xor_eq b: " << ( a xor_eq b ) << endl;
24
25  return 0;
26 )

```

输出结果:

```

    a and b: true
    a or b: true
    not a: false
a not_eq b: true
a bitand b: 22
a bit_or b: 22
    a xor b: 0
    compl a: -23
a and_eq b: 22
    a or_eq b: 3
a xor_eq b: 30

```

图 21.9 演示运算符关键字

程序声明和初始化两个整数 *a* 与 *b*, 用不同运算符关键字对 *a* 和 *b* 进行逻辑和位操作, 并输出每个操作的结果。

## 21.9 explicit 构造函数

第 8 章“运算符重载”中介绍过, 任何用一个参数调用的构造函数都可以让编译器进行隐式转换, 将构造函数接收的类型转换为构造函数定义的类对象。这种转换是自动进行的, 程序员不需要使用强制类型转换运算符。在有些情况中, 隐式转换是不合适的, 容易产生错误。例如, 图 8.4 中的 *Array* 类定义了一个取 *int* 参数的构造函数。这个构造函数的目的是生成一个 *Array* 对象, 包含几个用 *int* 参数指定个数的元素。但是这个构造函数可能让编译器错误地进行隐式转换。图 21.10 用第 8 章 *Array* 类的简化版本, 演示错误的隐式转换。

```

1 // Fig 21.10: array2.h
2 // Simple class Array (for integers)
3 #ifndef ARRAY1_H
4 #define ARRAY1_H
5

```

---

```

6 #include <iostream.h>
7
8 class Array {
9     friend ostream &operator<<( ostream &, const Array & );
10 public:
11     Array( int = 10 ); // default/conversion constructor
12     ~Array();          // destructor
13 private:
14     int size; // size of the array
15     int *ptr; // pointer to first element of array
16 };
17
18 #endif
19 // Fig 21.10: array2.cpp
20 // Member function definitions for class Array
21 #include <assert.h>
22 #include "array2.h"
23
24 // Default constructor for class Array (default size 10)
25 Array::Array( int arraySize )
26 {
27     size = ( arraySize > 0 ? arraySize : 10 );
28     cout << "Array constructor called for "
29          << size << " elements\n";
30
31     ptr = new int[ size ]; // create space for array
32     assert( ptr != 0 );    // terminate if memory not allocated
33
34     for ( int i = 0; i < size; i++ )
35         ptr[ i ] = 0;      // initialize array
36 }
37
38 // Destructor for class Array
39 Array::~Array() { delete [] ptr; }
40
41 // Overloaded output operator for class Array
42 ostream &operator<<( ostream &output, const Array &a )
43 {
44     int i;
45
46     for ( i = 0; i < a.size; i++ )
47         output << a.ptr[ i ] << ' ' ;
48
49     return output; // enables cout << x << y;
50 }
51 // Fig 21.10: fig21_10.cpp
52 // Driver for simple class Array
53 #include <iostream.h>
54 #include "array2.h"
55
56 void outputArray( const Array & );
57
58 int main()
59 {

```

```

60   Array integers1( 7 );
61
62   outputArray( integers1 );    // output Array integers1
63
64   outputArray( 15 );    // convert 15 to an Array and output
65
66   return 0;
67 }
68
69 void outputArray( const Array &arrayToOutput )
70 {
71     cout << "The array received contains:\n"
72           << arrayToOutput << "\n\n";
73 }

```

**输出结果：**

```

Array constructor called for 7 elements
The array received contains:
0 0 0 0 0 0 0

```

```

Array constructor called for 15 elements
The array received contains:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```

图 21.10 带有一个参数的构造函数与隐式转换

main 中第 60 行：

```
Array integers1( 7 );
```

定义 Array 对象 integers1 和调用使用一个参数的构造函数，该函数用 int 值 7 指定 Array 中的元素个数。我们修改 Array 构造函数，使其输出一行文本，表示调用 Array 构造函数和在 Array 中分配指定个数的元素。第 62 行：

```
outputArray( integers1 );    // output Array integers1
```

调用函数 outputArray（在第 69 行定义），输出 Array 的内容。函数 outputArray 接收 Array 的 const Array & 作为参数，然后用重载的流插入运算符 << 输出 Array。第 64 行：

```
outputArray( 15 );    // convert 15 to an Array and output
```

调用函数 outputArray，该函数用 int 值作 15 为参数。但没有取 int 参数的函数 outputArray，因此编译器检查 Array 类，确定是否有一个转换构造函数可以将 int 转换为 Array。由于 Array 类提供了一个转换构造函数，因此编译器用这个转换构造函数生成包含 15 个元素的临时 Array 对象，将临时 Array 对象传入函数 outputArray 以输出 Array。输出表明对 15 个元素的 Array 调用 Array 转换构造函数并输出 Array 内容。

C++ 提供关键字 explicit，通过转换构造函数取消隐式转换。声明为 explicit 的构造函数不能用于隐式转换。图 21.11 的程序演示 explicit 构造函数。

```

1 // Fig. 21.11: array3.h
2 // Simple class Array (for integers)

```

```

3 #ifndef ARRAY1_H
4 #define ARRAY1_H
5
6 #include <iostream.h>
7
8 class Array {
9     friend ostream &operator<<( ostream &, const Array & );
10 public:
11     explicit Array( int = 10 ); // default constructor
12     ~Array(); // destructor
13 private:
14     int size; // size of the array
15     int *ptr; // pointer to first element of array
16 };
17
18 #endif
19 // Fig. 21.11: array3.cpp
20 // Member function definitions for class Array
21 #include <assert.h>
22 #include "array3.h"
23
24 // Default constructor for class Array (default size 10)
25 Array::Array( int arraySize )
26 {
27     size = ( arraySize > 0 ? arraySize : 10 );
28     cout << "Array constructor called for "
29          << size << " elements\n";
30
31     ptr = new int[ size ]; // create space for array
32     assert( ptr != 0 ); // terminate if memory not allocated
33
34     for ( int i = 0; i < size; i++ )
35         ptr[ i ] = 0; // initialize array
36 }
37
38 // Destructor for class Array
39 Array::~~Array() { delete [] ptr; }
40
41 // Overloaded output operator for class Array
42 ostream &operator<<( ostream &output, const Array &a )
43 {
44     int i;
45
46     for ( i = 0; i < a.size; i++ )
47         output << a.ptr[ i ] << ' ';
48
49     return output; // enables cout << x << y;
50 }
51 // Fig. 21.11: fig21_11.cpp
52 // Driver for simple class Array
53 #include <iostream.h>
54 #include "array3.h"
55
56 void outputArray( const Array & );

```



```

57
58 int main()
59 {
60     Array integers1( 7 );
61
62     outputArray( integers1 );    // output Array integers1
63
64     outputArray( 15 );    // convert 15 to an Array and output
65
66     outputArray( Array( 15 ) ); // really want to do this!
67
68     return 0;
69 }
70
71 void outputArray( const Array &arrayToOutput )
72 {
73     cout << "The array received contains:\n"
74           << arrayToOutput << "\n\n";
75 }

```

**输出结果:**

```

Compiling...
Fig21_11.cpp
Fig21_11.cpp(14) : error: 'outputArray' :
    cannot convert parameter 1 from 'const int' to
    'const class Array &'
Array3.cpp

```

图 21.11 演示 explicit 构造函数

对图21.10程序所做的惟一修改是第11行增加关键字explicit, 声明用一个参数调用的构造函数。程序编译时, 编译器产生一个错误消息, 表示第64行传入outputArray的整数值不能转换为const Array &。编译器的错误消息在输出窗口中显示。第66行演示如何生成15个元素的Array 和用explicit 构造函数将其传入outputArray。

**常见编程错误 21.4**

试图调用声明为 explicit 的构造函数用于隐式转换是个语法错误。

**常见编程错误 21.5**

除了用一个参数调用的构造函数外, 对数据成员和成员函数使用 explicit 关键字是个语法错误。

**软件工程视点 21.5**

对于不需要编译器使用隐式转换的带有一个参数的构造函数使用 explicit 关键字。

## 21.10 mutable 类成员

第21.4节曾介绍过const\_cast运算符允许强制转换常量性。C++提供了一个存储类说明符mutable 代替const\_cast。即使在const 对象和const 成员函数中, mutable 数据成员也是可修改的。这样就不需要强制转换常量性。

**可移植性提示 21.2**

不管用 `const_cast` 还是用 C 语言式强制类型转换修改定义为常量的对象，其效果随编译器的不同而不同。

`mutable` 和 `const_cast` 都允许修改数据成员，但使用情况各不相同，对于不带 `mutable` 数据成员的 `const` 对象，运算符 `const_cast` 要在每次修改成员时使用。这样就可以大大减少意外修改成员的机会，因为成员并不总是可修改的。涉及 `const_cast` 的操作通常隐藏在成员函数的实现方法中。使用类的用户可能不知道正在修改成员。

**软件工程视点 21.6**

`mutable` 成员用于有“秘密”实现细节的类，这些细节不影响对象逻辑值。

图 21.12 的程序演示了 `mutable` 成员。

```
1 // Fig. 21.12: fig21_12.cpp
2 // Demonstrating storage class specifier mutable.
3 #include <iostream.h>
4
5 class TestMutable {
6 public:
7     TestMutable( int v = 0 ) { value = v; }
8     void modifyValue() const { value++; }
9     int getValue() const { return value; }
10 private:
11     mutable int value;
12 };
13
14 int main()
15 {
16     const TestMutable t( 99 );
17
18     cout << "Initial value: " << t.getValue();
19
20     t.modifyValue(); // modifies mutable member
21     cout << "\nModified value: " << t.getValue() << endl;
22
23     return 0;
24 }
```

**输出结果：**

```
Initial value: 89
Modified value: 100
```

图 21.12 演示 `mutable` 成员

程序在第 5 行定义 `TestMutable` 类，包含一个构造函数、两个函数和 `private mutable` 数据成员 `value`。第 8 行：

```
void modifyValue() const { value++; }
```

定义函数 `modifyValue` 为 `const` 函数，递增 `mutable` 数据成员 `value` 的值。通常，`const` 成员函数不修改数据成员，除非函数处理的对象（即 `this` 所指的对象）用 `const_cast` 转换为非 `const` 类型。由于 `value`

值是 mutable，因此这个 const 函数可以修改数据。第 9 行函数 getValue 是个 const 函数，返回一个 value。注意 getValue 可以改变 value，因为 value 是 mutable 类的成员。

第 16 行声明 const TestMutable 对象 t 并将其初始化为 99。第 18 行输出 value 的内容。第 20 行调用 const 成员函数 modifyValue，将 value 加 1。注意 t 和 modifyValue 都是 const 类型。第 21 行输出 value 值 (100)，证明实际修改了 mutable 数据类型。

## 21.11 类成员指针 (.\* 和 ->\*)

C++ 提供了访问类成员的.\* 和 ->\* 运算符，类成员指针与前面介绍的指针不同。试图通过指向类成员的指针使用 ->或.\* 运算符是个语法错误。图 21.12 演示了类成员指针运算符。

### 常见编程错误 21.6

试图通过指向类成员的指针使用 ->或.\* 运算符是个语法错误。

```
1 // Fig. 21.13 fig21_13.cpp
2 // Demonstrating operators .* and ->*
3 #include <iostream.h>
4
5 class Test {
6 public:
7     void function() { cout << "function\n"; }
8     int value;
9 };
10
11 void arrowStar( Test * );
12 void dotStar( Test * );
13
14 int main()
15 {
16     Test t;
17
18     t.value = 8;
19     arrowStar( &t );
20     dotStar( &t );
21     return 0;
22 }
23
24 void arrowStar( Test *tPtr )
25 {
26     void ( Test::*memPtr )() = &Test::function;
27     ( tPtr->*memPtr )(); // invoke function indirectly
28 }
29
30 void dotStar( Test *tPtr )
31 {
32     int Test::*vPtr = &Test::value;
33     cout << ( *tPtr ).*vPtr << endl; // access value
34 }
```

输出结果:

```
function
8
```

图 21.13 演示 \* 和 ->\* 运算符

程序声明 Test 类, 提供 public 成员函数 function 和 public 数据成员 value。函数 function 输出 “function”。第 11 行和 12 行建立函数 arrowStar 和 dotStar 的原型。第 16 行到 18 行实例化对象 t 并将 t 的数据成员 value 设置为 8。第 19 行和 20 行调用函数 arrowStar 和 dotStar, 每次调用传递 t 的地址。

第 26 行:

```
void ( Test::*memPtr )() = &Test::function;
```

在函数 arrowStar 中声明和初始化 memPtr 为类 Test 成员的指针, 该类成员是个无参数和返回 void 结果的函数。首先检查赋值语句左边, void 是成员函数的返回类型。空括号表示这个成员函数不取参数。中间括号指定指针 memPtr, 指向类 Test 的成员。Test::\* memPtr 周围的括号是必需的。注意, 如果不指定 Test::, 则 memPtr 是标准函数指针。下面要检查赋值语句右边。

#### 常见编程错误 21.7

声明成员函数指针而不将指针名放在括号中是个语法错误。

#### 常见编程错误 21.8

声明成员函数指针而不在指针名前面加上类名和作用域运算符 (::) 是个语法错误。

赋值语句右边用地址运算符 (&) 取得类中成员函数 function (不取参数并返回 void) 的偏移量。指针 memPtr 初始化为这个偏移量。注意第 26 行赋值的左边和右边都不引用特定对象。只有类名和作用域运算符 (::) 一起使用。如果没有 &Test::, 则第 26 行赋值语句右边是标准函数指针。

第 27 行:

```
( tPtr->*memPtr )();
```

调用 memPtr 所指的成员函数 (function)。->\* 运算符调用 memPtr 所指定偏移量处的成员函数。第 32 行:

```
int Test::*vPtr = &Test::value;
```

声明和初始化 vPtr 为 Test 类 int 数据成员的指针。赋值语句右边指定寻找数据成员 value 的偏移量。注意, 如果没有 Test::, 则 vPtr 是 int value 地址的 int \* 指针。

下一行:

```
cout << ( *tPtr ).*vPtr << endl;
```

用 \* 运算符访问 vPtr 所指的成员。注意, 在客户代码中, 我们只能对可访问成员使用类成员指针运算符。本例中, value 和 function 都是 public 类型的。在类的成员函数中, 类的所有成员都是可访问的。

#### 常见编程错误 21.9

在 \* 或 ->\* 的两个字符之间放上空格是个语法错误。

#### 常见编程错误 21.10

逆转 \* 和 ->\* 中的字符顺序是个语法错误。

## 21.12 多重继承与 virtual 基类

第9章介绍过多重继承，即一个类继承两个或几个类。例如，多重继承可以在C++标准库中形成 `istream` 类（如图 21.14）。

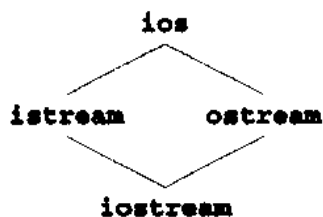


图 21.14 C++ 标准库中形成 `istream` 类的多重继承

`ios` 类是 `ostream` 和 `istream` 的基类，各形成一个单一继承，而 `iostream` 类从 `ostream` 和 `istream` 继承，使 `iostream` 类同时取得 `ostream` 和 `istream` 的功能。在多重继承层次中，图 21.14 的情况称为菱形继承。

由于 `ostream` 和 `istream` 都是从 `ios` 继承，因此 `iostream` 可能存在问题。`iostream` 类中可能包含重复子对象（即 `ostream` 和 `istream` 从 `ios` 继承的数据）。这个问题可能在 `iostream` 指针向上转换为 `ios` 指针时出现，即可能出现两个 `ios` 子对象。这时要用哪个子对象呢？这种情形是歧义的，会造成语法错误。程序 21.15 演示了这种歧义性，但通过隐式转换而不用向下转换，`iostream` 就不会遇到上述问题。本节介绍如何用 `virtual` 基类解决重复子对象的问题。

### 性能提示 21.1

重复子对象会浪费内存。

```
1 // Fig. 21.15: fig21_15.cpp
2 // Attempting to polymorphically call a function
3 // multiply inherited from two base classes.
4 #include <iostream.h>
5
6 class Base {
7 public:
8     virtual void print() const = 0; // pure virtual
9 };
10
11 class DerivedOne : public Base {
12 public:
13     // override print function
14     void print() const { cout << "DerivedOne\n"; }
15 };
16
17 class DerivedTwo : public Base {
18 public:
19     // override print function
20     void print() const { cout << "DerivedTwo\n"; }
21 };
22
23 class Multiple : public DerivedOne, public DerivedTwo {
```

```

24 public:
25     // qualify which version of function print
26     void print() const { DerivedTwo::print(); }
27 };
28
29 int main()
30 {
31     Multiple both;    // instantiate Multiple object
32     DerivedOne one;   // instantiate DerivedOne object
33     DerivedTwo two;   // instantiate DerivedTwo object
34
35     Base *array[ 3 ];
36     array[ 0 ] = &both;    // ERROR-ambiguous
37     array[ 1 ] = &one;
38     array[ 2 ] = &two;
39
40     // polymorphically invoke print
41     for ( int k = 0; k < 3; k++ )
42         array[ k ] -> print();
43
44     return 0;
45 )

```

#### 输出结果:

```

Compiling...
fig21_14.cpp
fig21_14.cpp(36) : error: '=' :
    ambiguous conversions from 'class Mutiple *' to
    'class Base *'

```

图 21.15 多态调用多重继承函数

程序定义 Base 类, 包含纯虚函数 print。类 DerivedOne 和 DerivedTwo 从 Base 通过 public 继承生成并重定义 print。类 DerivedOne 和 DerivedTwo 各包含一个 Base 子对象。

Multiple 类从 DerivedOne 和 DerivedTwo 多重继承。函数 print 重定义为调用 DerivedTwo 的 print。注意指定调用哪个子对象的限定符。

在 main 中, 生成层次中每个类的对象并声明 Base\* 指针的数组。每个数组元素初始化为该对象的地址。当 both 地址 (多重继承类型 Multiple) 隐式转换为 Base\* 时发生一个错误。both 对象包含从 Base 继承的重复子对象, 使调用 print 函数产生歧义。for 循环多态调用 array 所指的每个对象的 print。

这种重复子对象问题可以用 virtual 继承解决。基类用 virtual 继承时, 派生类中只出现一个子对象, 这个过程称为虚拟基类继承。图 21.16 的程序修改图 21.15, 使用 virtual 基类。

```

1 // Fig. 21.16: fig21_16.cpp
2 // Using virtual base classes.
3 #include <iostream.h>
4
5 class Base {
6 public:

```

```
7 // implicit default constructor
8
9 virtual void print() const = 0; // pure virtual
10 };
11
12 class DerivedOne : virtual public Base {
13 public:
14     // implicit default constructor calls
15     // Base default constructor
16
17     // override print function
18     void print() const { cout << "DerivedOne\n"; }
19 };
20
21 class DerivedTwo : virtual public Base {
22 public:
23     // implicit default constructor calls
24     // Base default constructor
25
26     // override print function
27     void print() const { cout << "DerivedTwo\n"; }
28 };
29
30 class Multiple : public DerivedOne, public DerivedTwo {
31 public:
32     // implicit default constructor calls
33     // DerivedOne and DerivedTwo default constructors
34
35     // qualify which version of function print
36     void print() const { DerivedTwo::print(); }
37 };
38
39 int main()
40 {
41     Multiple both; // instantiate Multiple object
42     DerivedOne one; // instantiate DerivedOne object
43     DerivedTwo two; // instantiate DerivedTwo object
44
45     Base *array[ 3 ];
46     array[ 0 ] = &both;
47     array[ 1 ] = &one;
48     array[ 2 ] = &two;
49
50     // polymorphically invoke print
51     for ( int k = 0; k < 3; k++ )
52         array[ k ] -> print();
53
54     return 0;
55 }
```

输出结果:

```
DerivedTwo  
DerivedOne  
DerivedTwo
```

图 21.16 使用 virtual 基础类

Base 类定义为包含纯虚函数 print。DerivedOne 类用下列语句从 Base 继承：

```
class DerivedOne : virtual public Base {
```

DerivedTwo 类用下列语句从 Base 继承：

```
class DerivedTwo : virtual public Base {
```

两个类都从 Base 继承，各包含一个 Base 子对象。Multiple 类从 DerivedOne 和 DerivedTwo 继承。Multiple 中只继承一个 Base 子对象。编译器允许发生转换（将 Multiple \* 变为 Base \*）。在 main 中，对层次中的每个类生成一个对象。Base 指针数组也已声明，每个 array 元素初始化为一个对象的地址。注意从 both 的地址已经可以向上转换到 Base \*。for 循环遍历 array 并多态调用每个对象的 print。

如果基类使用默认构造函数，则涉及 virtual 基类时的层次设计很简单。前两个例子中使用编译器生成的默认构造函数。如果 virtual 基类提供构造函数，则设计更复杂，因为最后派生类要负责初始化 virtual 基类。

本例中，Base、DerivedOne、DerivedTwo 和 Multiple 都是最后派生类。如果生成 Base 对象，则 Base 是最后派生类。如果生成 DerivedOne（或 DerivedTwo）对象，则 DerivedOne（或 DerivedTwo）是最后派生类。如果生成 Multiple 对象，则 Multiple 是最后派生类。不管类的层次多深，总有一个最后派生类，该派生类要负责初始化 virtual 基类。练习 21.17 中要让读者进行最后派生类的概念的练习。

#### 软件工程观点 21.7

对 virtual 基类使用默认构造函数能简化层次设计。

## 21.13 结束语

衷心希望你喜欢学习 C++ 和面向对象编程，愿读者坚持不懈地努力。

欢迎提供建议、批评、纠正和意见，我们将在下一版中感谢您。请向下列地址发送电子邮件：

deitel@deitel.com

祝你好运。

### 小结

- ANSI/ISO C++ 草案标准提供的 bool 数据类型可取值为 false 或 true，而旧式方法则用 0 表示 false，用非 0 表示 true。
- 流操纵算子 boolalpha 将输出流设置成将 bool 值显示为字符串“true”和“false”。
- ANSI/ISO C++ 草案标准引入 4 个新的强制类型转换运算符，优于 C 和 C++ 中使用的旧式强制类型转换。
- C++ 提供了在两种类型之间转换的 static\_cast 运算符。类型检查在编译时进行。
- const\_cast 运算符强制转换对象的常量性。



- C++ reinterpret\_cast 运算符用于非标准强制类型转换（如 void\* 校正为 int 等等）。
- 每个 namespace 定义一个全局标识符和全局变量范围。要使用 namespace 成员，就要用 namespace 名限定成员名，中间加上二元作用域运算符 (::) 或在使用这个名称之前用一个 using 语句。
- namespace 可以包含常量、数据、类、嵌套 namespace、函数等等。namespace 的定义应占有全局范围或其他 namespace 中的嵌套范围。
- 无名 namespace 成员占有全局 namespace。
- 运行时类型信息 (RTTI) 提供了运行时确定对象类型的方法。
- 编译时运算符 typeid 返回 type\_info 对象的引用。type\_info 对象是个系统维护的对象，表示一种类型。
- <typeinfo.h> 头文件 (ANSI/ISO C++ 草案标准中为 <typeinfo>) 定义 typeid。
- 运算符 dynamic\_cast 用于多态编程中保证在运行时发生正确的转换。对于无效转换操作，则 dynamic\_cast 的结果为 0。
- ANSI/ISO C++ 草案标准提供了运算符关键字 (如图 21.8)，可以代替几个 C++ 运算符。
- C++ 提供关键字 explicit，通过转换构造函数取消隐式转换。声明为 explicit 的构造函数不能用于隐式转换。
- 即使在 const 对象和 const 成员函数中，mutable 数据成员也是可修改的。
- C++ 提供了通过指针访问类成员的 \* 和 ->\* 运算符。
- 多重继承的重复子对象问题可以用 virtual 继承解决。基类用 virtual 继承时，派生类中只出现一个子对象，这个过程称为虚拟基类继承。

## 术语

.*	global variables 全局变量
->*	implicit conversion 隐式转换
anonymous namespace 匿名 namespace	most derived class 最后派生类
and	mutable
and_eq	name
bool	namespace
boolalpha	nested namespace 嵌套 namespace
bitand	not
bitor	not_eq
compl	operator keywords 运算符关键字
const_cast	or
diamond inheritance 菱形继承	or_eq
downcast 向下强制类型转换	pointer to class member operator 类成员指针运算符
dynamic_cast	pointer to data member 数据成员指针
explicit 显式	pointer to member function 成员函数指针
explicit conversion 显式转换	reinterpret_cast
false	
global namespace 全局 namespace	

RTTI (run-time type information)	运行时类型	typeid
信息		typeid
static_cast		using
subobject	子对象	virtual 虚拟
true	真	virtual base class 虚拟基类
typeid		xor
typeid		xor_eq

## 自测练习

### 21.1 填空：

- \_\_\_\_\_ 运算符用 namespace 限定成员。
- \_\_\_\_\_ 运算符可以强制转换对象的常量性。
- \_\_\_\_\_ 运算符可以在类型之间转换。

### 21.2 判断下列各题是否正确。如果不正确，请说明原因。

- 名字空间一定是惟一的。
- 名字空间不能有 namespace 成员。
- 数据类型 bool 是基础数据类型。

## 自测练习答案

21.1 a) 二元作用域 (::)。b) const\_cast。c) C 语言式、dynamic\_cast、static\_cast 或 reinterpret\_cast。

21.2 a) 不正确。程序员可能选择同名 namespace。

b) 不正确。名字空间可以嵌套。

c) 正确。

## 练习

### 21.3 填空：

- \_\_\_\_\_ 运算符确定运行时对象类型。
- \_\_\_\_\_ 关键字指定使用的 namespace 或 namespace 成员。
- 运算符 \_\_\_\_\_ 是逻辑或的运算符关键字。
- 存储说明符 \_\_\_\_\_ 允许修改 const 对象的成员。

### 21.4 判断下列各题是否正确。如果不正确，请说明原因。

- static\_cast 操作的有效性在编译时检查。
- dynamic\_cast 操作的有效性在运行时检查。
- typeid 是关键字。
- explicit 关键字适用于构造函数、成员函数和数据成员。

### 21.5 请说出下列表达式求值的结果（注意：有些表达式可能产生错误，请说明错误原因）。

- cout << false;
- cout << ( bool b = 8 )
- cout << ( a = true ); // a is of type int

- ```
d) cout << ( *ptr + true && p );    // *ptr is 10 and p is 8.88
e) // *ptr is 0 and m is false
    bool k = ( *ptr * 2 || ( true + 24 ) );
f) bool s = true + false;
g) cout << boolalpha << false << setw( 3 ) << true;
```
- 21.6 编写一个 namespace Currency, 定义常量成员 ONE、TWO、FIVE、TEN、TWENTY、FIFTY 和 HUNDRED。编写两个使用 Currency 的短程序, 一个程序利用所有常量, 其他程序只提供 FIVE。
- 21.7 编写一个程序, 用 reinterpret\_cast 运算符将不同指针类型转换为 int。是否有些转换会造成语法错误?
- 21.8 编写一个程序, 用 static\_cast 运算符将基础数据类型转换为 int。编译器是否允许转换为 int?
- 21.9 编写一个程序, 演示从派生类向上转换为基类, 用 static\_cast 运算符进行转换。
- 21.10 编写一个程序, 生成一个 explicit 构造函数, 取两个参数。编译器是否允许这样做? 删除 explicit 并进行隐式转换。编译器是否允许这样做?
- 21.11 explicit 构造函数有什么好处?
- 21.12 编写一个程序, 生成包含两个构造函数的类。一个构造函数取一个 int 参数, 另一个构造函数取一个 char\* 参数。编写一个驱动程序, 构造几个不同对象, 每个对象用不同类型传入构造函数。不用 explicit, 会发生什么情况, 只对取 int 的构造函数使用 explicit 时呢?
- 21.13 对下列 namespace, 判断下列各题是否正确, 并说明错误的原因。

```
1 #include <string>
2 namespace Misc {
3     using namespace std;
4     enum Countries { POLAND, SWITZERLAND, GERMANY,
5                     AUSTRIA, CZECH_REPUBLIC };
6     int kilometers;
7     string s;
8
9     namespace Temp {
10         short y = 77;
11         Car car;    // assume definition exists
12     }
13 }
14
15 namespace ABC {
16     using namespace Misc::Temp;
17     void *function( void *, int );
18 }
```

- a) 变量 y 可以在 namespace ABC 中访问。
- b) 对象 s 可以在 namespace Temp 中访问。
- c) 常量 POLAND 不可在 namespace Temp 中访问。
- d) 常量 GERMANY 可以在 namespace ABC 中访问。
- e) 函数 function 可以在 namespace Temp 中访问。

- f) 名字空间 ABC 中可以访问 Misc。
  - g) 对象 car 可以访问 Misc。
- 21.14 比较 mutable 和 const\_cast。至少用一个例子说明一个比另一个好。说明：这个练习不求编写任何代码。
- 21.15 编写一个程序，用 const\_cast 修改 const 变量。提示：用指针指向 const 标识符。
- 21.16 virtual 基类解决什么问题？
- 21.17 编写一个程序，使用 virtual 基类。层次顶上的类应提供至少取一个参数的构造函数（而不是提供默认构造函数）。这种情况下在继承层次中会出现什么问题？
- 21.18 寻找下列各题的错误，并说明如何纠正。
- a) namespace Name {  
    int x, y;  
    mutable int z;  
};
  - b) int integer = const\_cast< int > ( float );
  - c) namespace PCM( 111, "hello" );     // construct namespace
  - d) explicit int x = 99;

## 附录 A 运算符的优先级与结合律

运算符从上到下按优先级降低的顺序排列。

| 运算符                    | 类型             | 结合律  |
|------------------------|----------------|------|
| ::                     | 二元作用域          | 从左向右 |
| ::                     | 一元作用域          |      |
| ()                     | 括号             |      |
| []                     | 数组下标           |      |
| .                      | 通过对象选择成员       |      |
| ->                     | 通过指针选择成员       | 从右向左 |
| ++                     | 一元后置自增         |      |
| --                     | 一元后置自减         |      |
| typeid                 | 运行时类型信息        |      |
| dynamic_cast<type>     | 运行时类型检查的强制类型转换 |      |
| static_cast<type>      | 编译时类型检查的强制类型转换 |      |
| reinterpret_cast<type> | 非标准转换的强制类型转换   |      |
| const_cast<type>       | 对常量性进行强制类型转换   |      |
| ++                     | 一元前置自增         |      |
| --                     | 一元前置自减         |      |
| +                      | 一元正            | 从左向右 |
| -                      | 一元负            |      |
| !                      | 一元逻辑否定         |      |
| ~                      | 一元位取反          |      |
| ( type )               | C 语言式一元强制类型转换  |      |
| sizeof                 | 确定长度（字节数）      |      |
| &                      | 地址             |      |
| *                      | 复引用            |      |
| new                    | 动态内存分配         |      |
| new[ ]                 | 动态数组分配         |      |
| delete                 | 动态内存释放         |      |
| delete[ ]              | 动态数组删除         |      |
| .*                     | 通过对象的成员指针      | 从左向右 |
| ->*                    | 通过指针的成员指针      |      |
| *                      | 乘              | 从左向右 |
| /                      | 除              |      |
| %                      | 求模             |      |
| +                      | 加              | 从左向右 |
| -                      | 减              |      |
| <<                     | 位左移            | 从左向右 |
| >>                     | 位右移            |      |
| <                      | 小于             | 从左向右 |
| <=                     | 小于或等于          |      |
| >                      | 大于             |      |
| >=                     | 大于或等于          |      |

| (续表) |       |      |
|------|-------|------|
| 运算符  | 类型    | 结合律  |
| ==   | 等于    | 从左向右 |
| !=   | 不等于   |      |
| &    | 位与    | 从左向右 |
| ^    | 位异或   | 从左向右 |
|      | 位或    | 从左向右 |
| &&   | 逻辑与   | 从左向右 |
|      | 逻辑或   | 从左向右 |
| ?:   | 三元条件  | 从右向左 |
| =    | 赋值    | 从右向左 |
| +=   | 加法赋值  |      |
| -=   | 减法赋值  |      |
| *=   | 乘法赋值  |      |
| /=   | 除法赋值  |      |
| %=   | 求模赋值  |      |
| &=   | 位与赋值  |      |
| ^=   | 位异或赋值 |      |
| =    | 位或赋值  |      |
| <<=  | 位左移赋值 |      |
| >>=  | 位右移赋值 |      |
| ,    | 逗号    | 从左向右 |

图 A.1 运算符优先级表

## 附录 B ASCII 字符集

|    | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0  | nul | soh | stx | etx | eot | enq | ack | bel | bs  | ht  |
| 1  | nl  | vt  | ff  | cr  | so  | si  | dle | dcl | dc2 | dc3 |
| 2  | dc4 | nak | syn | etb | can | em  | sub | esc | fs  | gs  |
| 3  | rs  | us  | sp  | !   | "   | #   | \$  | %   | &   | '   |
| 4  | (   | )   | *   | +   | ,   | -   | .   | /   | 0   | 1   |
| 5  | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | :   | ;   |
| 6  | <   | =   | >   | ?   | @   | A   | B   | C   | D   | E   |
| 7  | F   | G   | H   | I   | J   | K   | L   | M   | N   | O   |
| 8  | P   | Q   | R   | S   | T   | U   | V   | W   | X   | Y   |
| 9  | Z   | [   | \   | ]   | ^   | _   | `   | a   | b   | c   |
| 10 | d   | e   | f   | g   | h   | i   | j   | k   | l   | m   |
| 11 | n   | o   | p   | q   | r   | s   | t   | u   | v   | w   |
| 12 | x   | y   | z   | {   |     | }   | ~   | del |     |     |

图 B.1 ASCII 字符集

该表左边的数是字符编码十进制值(0~127)的高位,表上边的数是字符编码的十进制值的低位。例如, 'F' 的字符代码是 70, '&' 的字符代码是 38。

注意: 本书大部分用户对用来表示英语字符的 ASCII 字符集感兴趣, 该字符集是表示各种语言字符的 Unicode 字符集的子集。关于 Unicode 字符集的信息, 参见下列 Web 站点:

<http://unicode.org>

## 附录C 数值系统

### 学习目的

- 了解基本数值系统的概念如基数、位值、符号值
- 了解如何运用以二进制、八进制和十六进制数值系统表示的数值
- 能够把二进制数简化为八进制或十六进制数
- 能够将八进制数和十六进制数转换为二进制数
- 能够进行十进制与二进制、八进制、十六进制数之间的转换
- 了解二进制算术运算及如何用补码表示负的二进制数

### C.1 简介

本附录介绍了C++程序员使用的重要的数值系统，特别是他们开发的项目要求不与“机器级”硬件交互时更是如此。这样的项目包括操作系统、计算机网络软件、编译器、数据库系统及各种高性能要求的应用程序。

当我们在C++程序中用到227或-63等等的整数时，我们假定了这些数值是以十进制数值系统（以10为基数）表示的。十进制数值系统中的数码有0、1、2、3、4、5、6、7、8和9。最小的数码为0，最大数码为9（比基数10小1）。计算机内部使用的是二进制数值系统（基数为2）。二进制数值系统只有两个数码，即0和1，最小数码为0，最大数码为1（比基数2小1）。

可以看到，二进制数比等值的十进制数长得多。汇编语言和像C这样的高级语言可以深入到“机器级”编程。但使用这些语言的程序员发现二进制数用起来很麻烦。因此，另外两种数值系统——八进制数值系统（基数为8）和十六进制数值系统（基数为16）因为能够方便地简化二进制数而受到了人们的欢迎。

八进制数值系统的数码范围为0-7。因为二进制数值系统和八进制数值系统都比十进制数值系统的数码少，所以它们的数码与相应的十进制数码相同。

而十六进制数值系统需要十六个数码——最小数码为0，最大数码的值相当于十进制数15（比基数16小1）。习惯上，我们用字母A到F表示对应于十进制数10~15的十六进制数码。这样在十六进制中我们可能得到仅含十进制数码的十六进制数如876，也可得到同时包含数码和字母的数值如8A55F以及只含字母的数值如FFE。有时十六进制数刚好拼成单词如FACE、FEED，这常使程序员感到别扭。

以上每种数值系统都用按位记数法——每个数位有一个不同的位值。例如：对于十进制数937（9、3、7称为符号值），我们说7在个位、3在十位、9在百位。注意每个数位实际是一个基数的乘幂（基数为10），其乘幂从右到左依次递增，从0开始，每次加1（见图C.3）。

再长的十进制数，后面将是千位（10的3次乘幂）、万位（10的4次乘幂）、十万位（10的5次乘幂）、百万位（10的6次乘幂）、千万位（10的7次乘幂），依次类推。



对于二进制数 101, 我们说最右边的 1 在一位, 0 在二位, 最左边的 1 在四位。注意, 每个数位实际是一个基数 2 的乘幂, 乘幂从右到左依次递增, 从 0 开始, 每次加 1 (见图 C.4)。

再长的二进制数, 后面将是八位 (2 的 3 次乘幂)、十六位 (2 的 4 次乘幂)、三十二位 (2 的 5 次乘幂)、六十四位 (2 的 6 次乘幂), 依次类推。

对八进制数 425, 我们说 5 在一位, 2 在八位, 4 在六十四位。每个数位实际是基数 8 的乘幂, 乘幂从右到左依次递增, 从 0 开始, 每次加 1 (见图 C.5)。

再长的八进制数, 后面将是五百二十位 (8 的 4 次乘幂)、三万二千七百六十八位 (8 的 5 次乘幂), 依次类推。

对十六进制数 3DA, 我们说 A 在个位, D 在十六位, 3 在二百五十六位。每个数位实际是基数 16 的乘幂, 乘幂从右到左依次递增, 从 0 开始, 每次加 1 (见图 C.6)。

再长的十六进制数, 后面将是四千零九十六位 (16 的 3 次乘幂)、三万二千七百六十八位 (16 的 4 次乘幂), 依次类推。

| 二进制数码 | 八进制数码 | 十进制数码 | 十六进制数码      |
|-------|-------|-------|-------------|
| 0     | 0     | 0     | 0           |
| 1     | 1     | 1     | 1           |
|       | 2     | 2     | 2           |
|       | 3     | 3     | 3           |
|       | 4     | 4     | 4           |
|       | 5     | 5     | 5           |
|       | 6     | 6     | 6           |
|       | 7     | 7     | 7           |
|       |       | 8     | 8           |
|       |       | 9     | 9           |
|       |       |       | A (十进制值 10) |
|       |       |       | B (十进制值 11) |
|       |       |       | C (十进制值 12) |
|       |       |       | D (十进制值 13) |
|       |       |       | E (十进制值 14) |
|       |       |       | F (十进制值 15) |

图 C.1 二进制、八进制、十进制和十六进制数值系统的数码

| 属性   | 二进制 | 八进制 | 十进制 | 十六进制 |
|------|-----|-----|-----|------|
| 基数   | 2   | 8   | 10  | 16   |
| 最小数码 | 0   | 0   | 0   | 0    |
| 最大数码 | 1   | 7   | 9   | F    |

图 C.2 二进制、八进制、十进制和十六进制数值系统的比较

| 十进制数值系统中的位值 |        |        |        |
|-------------|--------|--------|--------|
| 十进制数码       | 9      | 3      | 7      |
| 数位名         | 百位     | 十位     | 个位     |
| 位值          | 100    | 10     | 1      |
| 位值的乘幂形式     | $10^2$ | $10^1$ | $10^0$ |
| 底数 (10)     |        |        |        |

图 C.3 十进制数值系统中的位值

| 二进制数值系统中的位值 |       |       |       |
|-------------|-------|-------|-------|
| 二进制数码       | 1     | 0     | 1     |
| 数位名         | 四位    | 二位    | 一位    |
| 位值          | 4     | 2     | 1     |
| 位值的乘幂形式     | $2^2$ | $2^1$ | $2^0$ |
| 底数 (2)      |       |       |       |

图 C.4 二进制数值系统中的位值

| 八进制数值系统中的位值 |       |       |       |
|-------------|-------|-------|-------|
| 八进制数码       | 4     | 2     | 5     |
| 数位名         | 六十四位  | 八位    | 一位    |
| 位值          | 64    | 8     | 1     |
| 位值的乘幂形式     | $8^2$ | $8^1$ | $8^0$ |
| 底数 (8)      |       |       |       |

图 C.5 八进制数值系统中的位值

| 十六进制数值系统中的位值 |        |        |        |
|--------------|--------|--------|--------|
| 十六进制数码       | 3      | D      | A      |
| 数位名          | 二百五十六位 | 十六位    | 一位     |
| 位值           | 256    | 16     | 1      |
| 位值的乘幂形式      | $16^2$ | $16^1$ | $16^0$ |
| 底数 (16)      |        |        |        |

图 C.6 十六进制数值系统中的位值

C.2 将二进制数简化为八进制和十六进制数

八进制和十六进制数在计算的主要用途是简化冗长的二进制数。图 C.7 表明冗长的二进制数可用基数更高的数值系统简洁地表达。

| 十进制 | 二进制   | 八进制 | 十六进制 |
|-----|-------|-----|------|
| 0   | 0     | 0   | 0    |
| 1   | 1     | 1   | 1    |
| 2   | 10    | 2   | 2    |
| 3   | 11    | 3   | 3    |
| 4   | 100   | 4   | 4    |
| 5   | 101   | 5   | 5    |
| 6   | 110   | 6   | 6    |
| 7   | 111   | 7   | 7    |
| 8   | 1000  | 10  | 8    |
| 9   | 1001  | 11  | 9    |
| 10  | 1010  | 12  | A    |
| 11  | 1011  | 13  | B    |
| 12  | 1100  | 14  | C    |
| 13  | 1101  | 15  | D    |
| 14  | 1110  | 16  | E    |
| 15  | 1111  | 17  | F    |
| 16  | 10000 | 20  | 10   |

图 C.7 等值的十进制、二进制、八进制与十六进制数

八进制、十六进制系统与二进制系统有一个特别重要的关系，就是八进制与十六进制的基数（8和16）是二进制基数2的乘幂。考察下面的12位二进制数以及与之相等的八进制和十六进制数，看看这种关系是如何方便地把二进制数简化为八进制和十六进制数的。

|              |      |       |
|--------------|------|-------|
| 二进制数         | 八进制数 | 十六进制数 |
| 100011010001 | 4321 | 8D1   |

二进制转换为八进制，只需将12位二进制数按每三个连续位分为一组，并按以下方式将每组写成八进制数：

|     |     |     |     |
|-----|-----|-----|-----|
| 100 | 011 | 010 | 001 |
| 4   | 3   | 2   | 1   |

注意，在每组下面写的八进制数刚好与相应这三位二进制数相等（见图C.7）。

在从二进制到十六进制的转换中，我们可以发现同样的关系。它是将12位二进制数按照每4个连续位分成一组，并按以下方式将每组写成十六进制数：

|      |      |      |
|------|------|------|
| 1000 | 1101 | 0001 |
| 8    | D    | 1    |

可以看到，在每组下面写的十六进制数刚好与相应这四位二进制数相等（见图C.7）。

### C.3 将八进制和十六进制数转换为二进制数

上一节中，我们知道了如何将二进制数转换为等值的八进制和十六进制数，就是通过把二进制数位分组，然后简单地代之以等值的八进制或十六进制值。这个处理反过来也可以用于产生与给定八进制或十六进制数相等的二进制数。

例如，要将八进制数653转换为二进制数，只需将6、5和3分别写成与之相等的3位二进制数110、101和011，这就形成了与八进制653相等的9位二进制数110101011。

要将十六进制数FAD5转换为二进制数，只需将F、A、D和5分别写成与之相等的4位二进制数1111、1010、1101和0101，这就形成了与十六进制数FAD5相等的16位二进制数1111101011010101。

### C.4 将二进制、八进制和十六进制数转换为十进制数

因为我们已习惯于十进制，所以经常将二进制、八进制、十六进制数转换为十进制，从而找到“真实”值的感觉。C.1节中的图表示了十进制的位值。从其他进制转换为十进制，只需将每个数码的十进制数乘以其位值，然后求出这些积的和。例如，图C.8把二进制数110101转换成了十进制数53。

| 将二进制数转换为十进制数 |                                  |                    |                  |                  |                  |                  |
|--------------|----------------------------------|--------------------|------------------|------------------|------------------|------------------|
| 位值           | 32                               | 16                 | 8                | 4                | 2                | 1                |
| 符号值          | 1                                | 1                  | 0                | 1                | 0                | 1                |
| 乘积           | $1 \times 32 = 32$               | $1 \times 16 = 16$ | $0 \times 8 = 0$ | $1 \times 4 = 4$ | $0 \times 2 = 0$ | $1 \times 1 = 1$ |
| 总和           | $= 32 + 16 + 0 + 4 + 0 + 1 = 53$ |                    |                  |                  |                  |                  |

图C.8 将二进制数转换为十进制数

我们还可以用同样的办法将八进制数 7614 和十六进制数 AD3B 转换为十进制数 3980 和 44347 (见图 C.9 和 C.10)。

| 将八进制数转换为十进制数 |                               |                     |                  |                  |
|--------------|-------------------------------|---------------------|------------------|------------------|
| 位值           | 512                           | 64                  | 8                | 1                |
| 符号值          | 7                             | 6                   | 1                | 4                |
| 乘积           | $7 \times 512 = 3584$         | $6 \times 64 = 384$ | $1 \times 8 = 8$ | $4 \times 1 = 4$ |
| 总和           | $= 3584 + 384 + 8 + 4 = 3980$ |                     |                  |                  |

图 C.9 将八进制数转换为十进制数

| 将十六进制数转换为十进制数 |                                    |                       |                    |                   |
|---------------|------------------------------------|-----------------------|--------------------|-------------------|
| 位值            | 4096                               | 256                   | 16                 | 1                 |
| 符号值           | A                                  | D                     | 3                  | B                 |
| 乘积            | $A \times 4096 = 40960$            | $D \times 256 = 3328$ | $3 \times 16 = 48$ | $B \times 1 = 11$ |
| 总和            | $= 40960 + 3328 + 48 + 11 = 44347$ |                       |                    |                   |

图 C.10 将十六进制数转换为十进制数

## C.5 将十进制数转换为二进制、八进制或十六进制数

上一节中的转换是根据常规的按位记数法进行的。从十进制到二进制、八进制、十六进制的转换也遵循该规则。

例如将十进制数 57 转换为二进制。我们从右到左写出每列的位值，直到发现位值大于该十进制数的列。我们不需要该列，所以将之丢弃。这样首先得到：

位值: 64 32 16 8 4 2 1

然后，去掉位值为 64 的列，得到：

位值: 32 16 8 4 2 1

下一步，我们从左到右进行。先用 57 除以 32，得到商为 1，余数是 25，所以我们在 32 这一列下写上 1。然后用 25 除以 16，得到商为 1，余数是 9，所以我们在 16 这一列下写上 1。再用 9 除以 8，得到商为 1，余数是 1，所以我们在 8 这一列上写上 1。其后两列我们得到商为 0，所以写上两个 0。最后，1 除以 1 得 1，写上 1。这样就得到了下面的结果：

位值: 32 16 8 4 2 1  
符号值: 1 1 1 0 0 1

因此，十进制数 57 等于二进制数 111001。

将十进制数 103 转换成八进制。我们从右到左写每列的位值，直到发现位值大于该十进制数的列。我们不需要该列，所以将之丢弃。这样首先得到：

位值: 512 64 8 1

然后，我们去掉位值为 512 的列，得到：

位值: 64 8 1

下一步从左到右进行。用103除以64，得到商为1，余数是39，所以在64这一列下写上1。用39除以8，得到商为4，余数是7，所以在8这一列下写上4。最后，7除以1得1，没有余数，写上7。结果如下所示：

```
位值: 64 8 1
符号值: 1 4 7
```

因此，十进制数103等于八进制数147。

将十进制数375转换成十六进制。我们从右到左写每列的位值，直到发现位值大于该十进制数的列。我们不需要该列，所以将之丢弃。这样首先得到：

```
位值: 4096 256 16 1
```

然后，我们去掉位值为4096的列，得到：

```
位值: 256 16 1
```

下一步从左到右进行。用375除以256，得到商为1，余数是119，所以在256这一列下写上1。用119除以16，得到商为7，余数是7，所以在16这一列下写上7。最后，7除以1，没有余数，写上7。结果如下所示：

```
位值: 256 16 1
符号值: 1 7 7
```

因此，十进制数375等于十六进制数177。

## C.6 负的二进制数：补码表示法

本附录中的讨论都集中在正数上。这一节，我们解释计算机是如何用补码表示法表示负数。首先解释补码如何形成一个二进制数，然后看看为什么它能表示给定二进制数的负数。

考察一个32位整数的机器。假定：

```
int value = 13;
```

value的32位表达式是

```
00000000 00000000 00000000 00001101
```

要形成value的负值，我们首先利用C++的按位求反运算符(~)形成它的反码：

```
ones_Complement of Value= ~value
```

在机器内部，~value现在是对每个数位取反后的value，即0变成1，1变成0：

```
value:
00000000 00000000 00000000 00001101

~value (即 value 的反码):
11111111 11111111 11111111 11110010
```

要形成value的补码，我们只需将value的反码加1。即：

value 的补码:

```
11111111 11111111 11111111 11110011
```

如果它确实就等于 -13, 那么它与 13 的二进制数相加就应该得到结果 0, 如下所示:

```

00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----
00000000 00000000 00000000 00000000

```

最左边的进位被丢弃, 事实上我们得到了结果 0。如果我们将一个数的反码加上这个数, 结果将是所有位全部为 1。要得到全零的方法就是用反码加 1 得到补码。加 1 导致每位变为 0 并进 1。进位一直左移直到被最左位丢弃, 故结果数为全零。

计算机执行减法:

```
x = a - value;
```

实际上是执行 a 加 value 的补码:

```
x = a + (~value + 1);
```

假定 a 为 27, value 为 13。如果 value 的补码确实是 value 的负数, 那么 value 的补码加 a 的结果应该是 14, 如下所示:

```

a (=27)          00000000 00000000 00000000 00011011
+ (~value+1)    + 11111111 11111111 11111111 11110011
-----
00000000 00000000 00000000 00001110

```

结果的确等于 14。

## 小结

- 当我们在 C++ 程序中用到 227 或 -63 等等的整数时, 这些值是自动假定是以十进制数值系统 (以 10 为基数) 表示的。十进制数值系统中的数码有 0、1、2、3、4、5、6、7、8 和 9。最小的数码为 0, 最大数码为 9 (比基数 10 小 1)。
- 计算机内部使用的是二进制数值系统 (基数为 2)。二进制数值系统只有两个数码, 即 0 和 1, 最小数码为 0, 最大数码为 1 (比基数 2 小 1)。
- 因为八进制 (基数为 8) 和十六进制 (基数为 16) 数值系统可方便地简化二进制数, 所以被广为应用。
- 八进制数值系统的数码是从 0~7。
- 十六进制数值系统需要十六个数码, 最小数码为 0, 最大数码的值相当于十进制数 15 (比基数 16 小 1)。习惯上, 我们用字母 A 到 F 表示对应于十进制数 10~15 的十六进制数码。
- 每一种数值系统都使用了按位记数法。数码所在的每一位有不同的位值。
- 八进制、十六进制数值系统与十进制数值系统之间有一个特别重要关系, 就是八进制和十六进制的基数 (8 和 16) 是二进制基数 (2) 的乘幂。
- 只需将八进制数的每个数位用相等的三位二进制数表示就可以把八进制数转换为二进制数。
- 只需将十六进制数的每个数位用相等的四位二进制数表示就可以把十六进制数转换为二进制数。

- 因为我们习惯于十进制，所以经常将二进制、八进制或十六进制数转换为十进制，以获得“真实”值的感受。
- 从其他进制转换为十进制，只需将每个数码的十进制数乘以其位值，然后求出这些积的和。
- 计算机是用补码表示法表示负数。
- 在二进制中，要形成一个值的负值，首先用C语言的按位取反运算符(~)形成它的反码(使得该值的所有位取反)，然后只需将该值的反码加1。

## 术语

|                                           |                                    |
|-------------------------------------------|------------------------------------|
| base 基数                                   | digit 数码                           |
| base 2 number system 以2为基数的数值系统           | hexadecimal number system 十六进制数值系统 |
| base 8 number system 以8为基数的数值系统           | negative value 负值                  |
| base 10 number system 以10为基数的数值系统         | octal number system 八进制数值系统        |
| base 16 number system 以16为基数的数值系统         | one's complement notation 反码表示法    |
| binary number system 二进制数值系统              | positional notation 按位记数法          |
| bitwise complement operator(~) 按位取反运算符(~) | positional value 位值                |
| conversions 转换                            | symbol value 符号值                   |
| decimal number system 十进制数值系统             | two's complement notation 补码表示法    |

## 自测练习

- C.1 十进制、二进制、八进制和十六进制数值系统的基数分别是 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- C.2 (选择) 通常，一个给定二进制数的十进制、八进制和十六进制表示法含有比二进制更(多/少)的数位。
- C.3 (判断正误) 用十进制数值系统的通常原因是它通过将四位二进制组替换为十进制数位，简化了二进制数。
- C.4 (选择) 一个非常大的二进制数的(八进制/十六进制/十进制)表示法是最简结的。
- C.5 (判断正误) 任何进制中的最大数码都是比基数大1。
- C.6 (判断正误) 任何进制中的最小数码都是比基数小1。
- C.7 不论是在二进制、八进制、十进制还是在十六进制中，任何数的最右边数位的位值通常等于\_\_\_\_\_。
- C.8 不论是在二进制、八进制、十进制还是在十六进制中，任何数的次右边数位的位值通常等于\_\_\_\_\_。
- C.9 补齐以下表示各数值系统的右边4个位值。

|      |      |     |     |     |
|------|------|-----|-----|-----|
| 十进制  | 1000 | 100 | 10  | 1   |
| 十六进制 | ...  | 256 | ... | ... |
| 二进制  | ...  | ... | ... | ... |
| 八进制  | 512  | ... | 8   | ... |

- C.10 将二进制数 110101011000 转换为八进制数和十六进制数。  
 C.11 将十六进制数 FACE 转换为二进制数。  
 C.12 将八进制数 7316 转换为二进制数。  
 C.13 将十六进制数 4FEC 转换为八进制数。提示：先转换为二进制数。  
 C.14 将二进制数 1101110 转换为十进制数。  
 C.15 将八进制数 317 转换为十进制数。  
 C.16 将十六进制数 EFD4 转换为十进制数。  
 C.17 将十进制数 177 转换为二进制、八进制和十六进制数。  
 C.18 写出把十进制数 417 表示为二进制数的过程。然后写出其反码和补码。  
 C.19 一个数的补码加上其本身会是什么结果？

### 自测练习答案

- C.1 10、2、8、16  
 C.2 少。  
 C.3 不正确。  
 C.4 十六进制。  
 C.5 不正确。任何进制中的最大数码都是比基数小 1。  
 C.6 不正确。任何进制中的最小数码都是 0。  
 C.7 1 (基数的 0 次幂)。  
 C.8 该数值系统的基数。
- |     |      |      |     |    |   |
|-----|------|------|-----|----|---|
| C.9 | 十进制  | 1000 | 100 | 10 | 1 |
|     | 十六进制 | 4096 | 256 | 16 | 1 |
|     | 二进制  | 8    | 4   | 2  | 1 |
|     | 八进制  | 512  | 64  | 8  | 1 |
- C.10 八进制数 6530；十六进制数 D58。  
 C.11 二进制数 1111 1010 1100 1110。  
 C.12 二进制数 111 011 001 110。  
 C.13 二进制数 0 100 111 111 101 100；八进制数 47754。  
 C.14 十进制数  $2+4+8+32+64=110$ 。  
 C.15 十进制数  $7+1*8+3*64=7+8+192=207$ 。  
 C.16 十进制数  $4+13*16+15*256+14*4096=61396$ 。  
 C.17 十进制数 177 转换成二进制数：
- |                                                 |     |    |    |    |   |   |   |   |
|-------------------------------------------------|-----|----|----|----|---|---|---|---|
| 256                                             | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 128                                             | 64  | 32 | 16 | 8  | 4 | 2 | 1 |   |
| $(1*128)+(0*64)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)$ |     |    |    |    |   |   |   |   |
| 10110001                                        |     |    |    |    |   |   |   |   |
- 转换成八进制数：
- |                      |    |   |   |
|----------------------|----|---|---|
| 512                  | 64 | 8 | 1 |
| 64                   | 8  | 1 |   |
| $(2*64)+(6*8)+(1*1)$ |    |   |   |
| 261                  |    |   |   |
- 转换成十六进制数：
- |     |    |   |
|-----|----|---|
| 256 | 16 | 1 |
|-----|----|---|



16 1  
 $(11 \times 16) + (1 \times 1)$   
 $(B \times 6) + (1 \times 1)$   
 B1

C.18 二进制:

512 256 128 64 32 16 8 4 2 1  
 256 128 64 32 16 8 4 2 1  
 $(1 \times 256) + (1 \times 128) + (0 \times 64) + (1 \times 32) + (0 \times 16) + (0 \times 8) + (0 \times 4) + (0 \times 2) + (1 \times 1)$   
 110100001

反码: 001011110

补码: 001011111

检验: 原二进制数加它的补码。

110100001  
 001011111  
 —————  
 00000000

C.19 为0。

## 练习

C.20 有人说在12进制数值系统中我们的许多计算会更容易, 因为12比十进制的基数10更具有可除性。在12进制中, 什么是最小数码? 什么可能是12进制数位的最高表示符? 其最右边四位的位值是多少?

C.21 任何数值系统中数的次右边数位的位值相对应的最高符号值是什么?

C.22 补齐以下各数值系统的右边4个位值。

|      |      |     |     |     |
|------|------|-----|-----|-----|
| 十进制  | 1000 | 100 | 10  | 1   |
| 6进制  | ...  | ... | 6   | ... |
| 13进制 | ...  | 169 | ... | ... |
| 3进制  | 27   | ... | ... | ... |

C.23 将二进制数100101111010转换成八进制和十六进制数。

C.24 将十六进制数3A7D转换成二进制数。

C.25 将十六进制数765F转换成八进制数。提示: 先转换成二进制。

C.26 将二进制数1011110转换成十进制数。

C.27 将八进制数426转换成十进制数。

C.28 将十六进制FFFF转换成十进制数。

C.29 将十进制数299转换成二进制、八进制、十六进制数。

C.30 写出把十进制数779转换成二进制数的过程。然后写出其反码和补码。

C.31 一个数的补码加上其本身会是什么结果?

C.32 写出在32位整数的机器上整数值-1的补码。

## 附录 D 有关 C++ 的 Internet 与 Web 资源

(说明:本附录是由 Abbey Deitel 编写的,她是 Carnegie Mellon 大学工业管理专业的毕业生)。下面列出一些宝贵的有关 C++ 的 Internet 和 Web 资源,包括常见问题、教程、如何取得 ANSI/ISO C++ 草案标准、流行的 C++ 编译器信息和如何获得可下载的编译器、演示例程、书籍、教程、软件工具、文章、访谈、会议资料、刊物与杂志、联机课程、新闻组与职业资源。

关于 American National Standards Institute 与购买标准文档的信息,见 <http://www.ansi.org>。

### D.1 资源

<http://www.progsources.com/index.html>

Programmer 的 Source 是许多编程语言(包括 C++)的宝贵信息资源。其中列出了工具、编译器、软件、书籍和其他 C++ 资源。

<http://www.intranet.ca/~sshah/booklist.html#C++>

Programmer 的 Book List 中列出了 30 多本 C++ 书籍。

<http://www.vcdj.com/default.htm>

Visual C++ Developer 的 Journal 主页提供最新文章、会议清单、作业库、演示例程与资源。

[http://webster.ucr.edu/Page\\_cpp/resource.html](http://webster.ucr.edu/Page_cpp/resource.html)

C 和 C++ 资源页面链接一些 C++ 相关的 Web 站点。

<http://www.genitor.com/resources.htm>

Developer Resources 站点链接 C++ 编译器、重要 C++ 工具、C/C++ Users Journal 源代码和出版物。

<http://www.possibility.com/Cpp/CppCodingStandard.html>

C++ Coding Standard 站点有大量关于 C++ 编程语言的信息和 Web 上的 C++ 资源清单。

<http://help-site.com/cpp.html>

Help-site.com 提供 Web 上的 C++ 资源链接。

### D.2 教程

<http://info.desy.de/gna/html/cc/index.html>

可以下载 Introduction to Object-Oriented Programming Using C++ 教程,也可以注册 Web 课程。其中有面向对象编程和 C++ 编程语言方面的推荐书目。

<http://uu-gna.mit.edu:8001/uu-gna/text/cc/Tutorial/tutorial.html>

Introduction to Object-Oriented Programming Using C++教程分为10章,各有一组练习和习题解答。

<http://www.icce.rug.nl/docs/cplusplus/cplusplus.html>

这个教程由大学教授编写,帮助C语言程序员学习C++编程。

<http://www.rdw.tec.mn.us/>

Red Wing/Winona Technical College提供可计入学分的联机C++课程。

<http://www.zdu.com/zdu/catalog/programming.htm>

ZD Net University提供大量与C++编程语言有关的联机课程。

<http://library.advanced.org/3074/>

这个教程帮助Pascal语言程序员学习C++编程。

<http://hyperion.advanced.org/11742/home.htm>

这个教程帮助DOS或Windows平台上的程序员学习高级课程,但没有目录表,无法评估教程中的所有信息。

<http://www.swcp.com/~dodrill/cppdoc/cpplist.htm>

这是C语言程序员学习C++编程的教程,包含12章内容、例子、源代码和练习解答。

<ftp://rtfm.mit.edu/pub/usenet/news.answers/C-ftq/learn-c-cpp-today>

这个站点有一系列C++教程及各自的说明,还有C与C++编程语言起源的信息,以及不同平台上不同C++编译器的信息。

### D.3 常见问题 (FAQ)

<http://reality.sgi.com/austern/std-c++/faq.html>

这个FAQ站点针对C++ ANSI/ISO草案标准、C++编程语言设计和最近更改语言和标准的信息。

<http://www.cis.ohio-state.edu/hypertext/faq/bngusenet/comp/lang/c++/top.html>

这个站点列出许多C++常见问题。

<http://www.trmphrst.demon.co.uk/cpplibs1.html>

这是个C++库的FAQ,其中有关于C++标准库的大量常见问题。

<http://www.up.ac.za/information/c-c++-learn.html>

这是C与C++编程的优秀资源,有多种C/C++教程和相关FAQ站点链接。

<http://pneuma.phys.ualberta.ca/~burris/cpp.htm>

Internet Link Exchange是C++信息的另一宝库,链接与comp.lang.c++、C++标准库和MFC相关的FAQ。

<http://www.math.uio.no/nett/faq/C-faq/faq.html>

comp.lang.c++ 列出常见问题及答案。

[http://lglwww.epfl.ch/~wolf/c\\_std.html](http://lglwww.epfl.ch/~wolf/c_std.html)

一系列 C++ 编程语言 ISO 标准的 FAQ。

[http://www.ses.com/~clarke/C++FAQ\\_Book.html](http://www.ses.com/~clarke/C++FAQ_Book.html)

这个站点有一系列 C++ 常见问题及答案。

<http://www.cerfnet.com/~mpcline/C++-FAQs-Lite/>

这个站点的常见问题及答案分为 35 类，有丰富的 C++ 信息，有些答案中包含代码例子。

[http://altair.chem-eng.kyushu-u.ac.jp/public\\_html/takeda/faq/C++\\_FAQ.html](http://altair.chem-eng.kyushu-u.ac.jp/public_html/takeda/faq/C++_FAQ.html)

这个站点有丰富的 C++ 常见问题及答案，目录表将问题分为 33 类，便于查找。

## D.4 comp.lang.c++

<http://weblab.research.att.com/phoaks/comp/lang/c++/resources0.html>

这个站点有大量与 comp.lang.c++ 有关的资源。页面标题“People Helping One Another Know Stuff”说明了这个站点的内容。其中有 40 多个 C++ 信息资源。

<http://www.r2m.com/windev/cpp-compiler.html>

这是 C++ 开发人员的宝贵资源，有 20 多个 C++ 相关站点链接。

<http://home.istar.ca/~stepanv/>

这个站点列出 25 个 C++ 编程的相关文章与信息站点的链接。其中列出的课程包括面向对象图形、ANSI C++ 草案标准、标准模板库、MFC 资源、OWL 资源、C++ 虚拟库和教程。

<http://kom.net/~dbrick/newspage/comp.lang.c++.html>

这个站点链接与 comp.lang.c++ 层次相关的新闻组。

<http://www.austinlinks.com/CPlusPlus/>

Quadralay Corporation 的 Web 站点链接 C++ 资源，包括 Visual C++/MFC Libraries、C++ 编程信息、C++ 职业资源、一系列教程和其他 C++ 学习辅助工具。

[http://db.csie.ncu.edu.tw/~kant\\_c/C/chapter2\\_21.html](http://db.csie.ncu.edu.tw/~kant_c/C/chapter2_21.html)

这个 Web 站点列出 ANSI C 标准库函数。

[http://wwwcn1.cern.ch/asd/geant/geant4\\_public/coding\\_standards/coding/coding\\_2.html](http://wwwcn1.cern.ch/asd/geant/geant4_public/coding_standards/coding/coding_2.html)

丰富的 C++ 标准信息资源。

<http://cuiwww.unige.ch/OSG/Vitek/Compilers/Year86/msg00046.html>

“The C standard on segmented machines”

<http://www.csci.csusb.edu/dick/c++std/>

这个站点链接ANSI/ISO C++ 草案标准与Usenet新闻组comp.std.c++，提供有关标准的新信息。

<http://ibd.ar.com/ger/comp.lang.c++.html>

Green Eggs Report 列出100多个comp.lang.c++方面的URL。

<http://www.ts.umu.se/~maxell/C++/>

这个站点提供一些C++类的代码例子。

<http://www.quadralay.com/Cplusplus/>

这是C++编程信息、学习C++、C++职业资源和其他C++相关信息的重要资源。

<http://www.research.att.com/~bs/homepage.html>

这是Bjarne Stroustrup的主页，他是C++编程语言设计者，提供了一系列C++资源、FAQ和其他有用的C++信息。

<http://www.cygnum.com/misc/wp/draft/index.html>

这个站点有ANSI/ISO C++ 工作草案的HTML文本。

<http://www.cs.bham.ac.uk/~jdm/cpp.html>

这是所遇到的最佳的C++资源，连接FAQ清单、Web C++课程、编译器信息、STL链接、C++库、C++编程工具和C++扩展。

<http://www.austinlinks.com/Cplusplus/>

这个站点包括ANSI/ISO C++、C++编程信息、C++职业资源、一系列教程和其他C++学习辅助工具。

<ftp://research.att.com/dist/c++std/WP/CD2/>

这个站点有当前的ANSI/ISO C++ 草案标准。

news:comp.lang.c++

面向对象的C++语言。

news:comp.lang.c++.leda

LEDA库的各个方面。

news:comp.lang.c++.moderated

C++语言技术讨论。

## D.5 编译器

<http://www.progsources.com/index.html>

Programmer的Source是许多编程语言（包括C++）的宝贵信息资源。其中列出工具、编译器、软件、书籍和其他C++资源。编译器清单按平台列出。

<http://www.cygnum.com/misc/gnu-win32/>

Cygnus Web 站点免费提供 GNU 开发环境。

<http://www.remcomp.com/lcc-win32/>

从这个 Web 站点可以免费下载 Windows 95/NT 的 LCC-Win32 编译器。

<http://www.microsoft.com/visualc/>

Microsoft Visual C++ 主页提供 Microsoft Visual C++ 编译器的产品信息、概述、补充材料和订购信息。

<http://www.powersoft.com/products/languages/watccpl.html>

Powersoft 公司关于 Watcom C/C++ version 11.0 的产品新闻和信息。这个编译器无法从 Web 站点下载，但提供了订购信息。

<http://www.borland.com/borlandcpp/cppprod.html>

Borland C++ 5.0 Development Suite 演示程序可以免费下载，并提供了订购信息。

<http://netserv.borland.com/borlandcpp/cppcomp/turbocpp.html>

Borland Turbo C++ Visual Edition for Windows 编译器的 Web 站点。

[http://www.symantec.com/scpp/fs\\_scpp72\\_95.html](http://www.symantec.com/scpp/fs_scpp72_95.html)

Windows 95 与 Windows NT 的 Symantec C++ 7.5。

[http://www.software.ibm.com/ad/visualage\\_c++/windows/vacwin.html](http://www.software.ibm.com/ad/visualage_c++/windows/vacwin.html)

IBM VisualAge for C++ for Windows 的主页。

[http://www.software.ibm.com/ad/visualage\\_c++/os2/](http://www.software.ibm.com/ad/visualage_c++/os2/)

IBM VisualAge for C++ for OS/2 Version 3.0。

<http://www.metrowerks.com/products/>

Metrowerks CodeWarrior for Macintosh 或 Windows。

## D.6 开发工具

<http://www.quintessoft.com/>

Quintessoft Engineering 公司提供的 Code Navigator for C++ 是 Windows 95/NT 上的 C++ 开发工具。从该站点可以得到一些产品信息，客户意见、免费试用版下载和价格信息。

## 参考文献

- ( A192 ) Allison, C., "Text Processing I, " *The C Users Journal*, Vol.10, No.10, October 1992, pp.23-28.
- ( A192a ) Allison, C., "Text Processing II, " *The C Users Journal*, Vol.10, No.12, December 1992, pp.73-77.
- ( A193 ) Allison, C., "Code Capsules: A C++ Date Class, Part I, " *The C Users Journal*, Vol.11, No.2, February 1993, pp.123-131.
- ( A194 ) Allison, C., "Conversions and Casts, " *The C/C++ Users Journal*, Vol.12, No.9, September 1994, pp.67-85.
- ( Am95 ) Almarode, J., "Object Security, " *Smalltalk Report*, Vol.5, No.3 November-December 1995, pp.15-17.
- ( An90 ) ANSI, *American National Standard for Information Systems-Programming Language C ( ANSI Document ANSI/ISO 9899:1990 )*, New York, NY:American National Standards Institute, 1990.
- ( An94 ) *American National Standard, Programming Language C++*. [Approval and technical development work is being conducted by Accredited Standards Committee X3, Information Technology and its Technical Committee X3J16, Programming Language C++, respectively. For further details, contact X3 Secretariat, 1250 Eye Street, NW, Washington, DC 20005.]
- ( An92 ) Anderson, A.E. and W.J.Heinze, *C++ Programming and Fundamental Concepts*, Englewood Cliffs, NJ:Prentice Hall, 1992.
- ( Ba92 ) Baker, L., *C Mathematical Function Handbook*, New York, NY:McGraw Hill, 1992.
- ( Ba93 ) Bar-David, T., *Object-Oriented Design for C++*, Englewood Cliffs, NJ:Prentice Hall, 1993.
- ( Be94 ) Beck, K., "Birds, Bees and Browsers-Obvious Sources of Object, " *The Smalltalk Report*, Vol.3, No.8 June 1994, p.13.
- ( Be93a ) Becker, P., "Conversion Confusion, " *C++ Report*, October 1993, pp.26-28.
- ( Be93 ) Becker, P., "Shrinking the Big Switch Statement, " *Windows Tech Journal*, Vol.2, No.5, May 1993, pp.26-33.
- ( Bd93 ) Berard, E.V., *Essays on Object Oriented Software Engineering: Volume I*, Englewood Cliffs, NJ:Prentice Hall, 1992.
- ( Bi95 ) Binder, R.V., "State-Based Testing, " *Object Magazine*, Vol. 5, No.4, August 1995, pp.75-78.
- ( Bi95a ) Binder, R.V., "State-Based Testing: Sneak Paths and Conditional Transitions, " *Object Magazine*, Vol.5, No.6, October 1995, pp.87-89.

- ( Bl92 ) Blum, A., *Neural Networks in C++: An Object-Oriented Framework for Building Connectionist Systems*, New York, NY: John Wiley & Sons, 1992.
- ( Bo91 ) Booch, G., *Object-Oriented Design with Applications*, Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991.
- ( Bo94 ) Booch, G., *Object-Oriented Analysis and Design*, Second Edition, Reading, MA: Addison-Wesley Publishing Company, 1994.
- ( Bo96 ) Booch, G., *Object Solutions*, Reading, MA: Benjamin/Cummings, 1996.
- ( Ca92 ) Cargill, T., *Programming Style*, Reading, MA: Addison-Wesley Publishing Company, 1992.
- ( Ca95 ) Carroll, M.D. and M.A.Ellis, *Designing and Coding Reusable C++*, Reading, MA: Addison-Wesley Publishing Company, 1995.
- ( Co95 ) Coplien, J.O. and D.C.Schmidt, *Pattern Languages of Program Design*, Reading, MA: Addison-Wesley Publishing Company, 1995.
- ( C++97 ) X3 Secretariat: *Draft Standard-The C++ Language*. X3J16/97-14882. Information Technology Council ( NSITC ), Washington, DC, USA: 1997.
- ( De90 ) Deitel, H.M., *Operating Systems*, Second Edition, Reading, MA: Addison-Wesley, 1990.
- ( De94 ) Deitel, H.M. and P.J.Deitel, *C How to Program* ( Second Edition ), Englewood Cliffs, NJ: Prentice Hall, 1994.
- ( De98 ) Deitel, H.M. and P.J.Deitel, *Java How to Program*, Second Edition, Upper Saddle River, NJ: Prentice Hall, 1998.
- ( De98a ) Deitel, H.M. and P.J.Deitel, *The Java Multimedia Cyber Classroom*, Second Edition, Upper Saddle River, NJ: Prentice Hall, 1998.
- ( Du91 ) Duncan, R., "Inside C++: Friend and Virtual Functions, and Multiple Inheritance." *PC Magazine*, Vol.10, No.17, October 15, 1991, pp.417-420.
- ( El90 ) Ellis, M.A. and B.Stroustrup, *The Annotated C++ Reference Manual*, Reading, MA: Addison-Wesley, 1990.
- ( Em92 ) Embley, D.W.; B.D.Kurtz; and S.N.Woodfield, *Object-Oriented Systems Analysis*, Englewood Cliffs, NJ: Yourdon Press, 1992.
- ( En90 ) Entsminger, G., *The Tao of Objects: A Beginner's Guide to Object-Oriented Programming*, Redwood City, CA: M&T Books, 1990.
- ( FL93 ) Flamiig, B., *Practical Data Structures in C++*, New York, NY: John Wiley & Sons, 1993.
- ( Ga95 ) Gamma, E.; R. Helm; R.Johnson; and J.Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley Publishing Company, 1995.
- ( Ge89 ) Gehani, N., and W.D.Roome, *The Concurrent C Programming Language*, Summit, NJ: Silicon Press, 1989.
- ( Gi92 ) Giancola, A. and L.Baker, "Bit Arrays with C++," *The C Users Journal*, Vol.10, No.7, July, 1992, pp.21-26.
- ( Gl95 ) Glass, G. and B.Schuchert, *The STL<Primer>*, Upper Saddle River, NJ: Prentice Hall PTR, 1995.



- (Go95) Gooch, T., "Obscure C++" *Inside Microsoft Visual C++*, Vol.6, No.11, November 1995, pp.13-15.
- (Ha90) Hansen, T.L., *The C++ Answer Book*, Reading, MA: Addison-Wesley, 1990.
- (He97) Henricson, M. and E.Nyquist, *Industrial Strength C++: Rules and Recommendations*, Upper Saddle River, NJ: Prentice Hall, 1997.
- (Ja93) Jacobson, I., "Is Object Technology Software's Industrial Platform?" *IEEE Software Magazine*, Vol.10, No.1, January 1993, pp.24-30.
- (Ja89) Jaeschke, R., *Portability and the C Language*, Indianapolis, IN: Hayden Books, 1989.
- (Ke88) Kernighan, B.W., and D.M.Ritchie, *The C Programming Language* (Second Edition), Englewood Cliffs, NJ: Prentice Hall, 1988.
- (Kn92) Knight, A., "Encapsulation and Information Hiding," *The Smalltalk Report*, Vol.1, No.8 June 1992, pp. 19-20.
- (Ko90) Koenig, A. and B. Stroustrup, "Exception Handling for C++ (revised)," *Proceedings of the USENIX C++ Conference*, San Francisco, CA, April 1990.
- (Ko91) Koenig, A., "What is C++ Anyway?" *Journal of Object-Oriented Programming*, April/May 1991, pp.48-52.
- (Ko94) Koenig, A., Implicit Base Class Conversions, " *C++ Report*, Vol.6, No.5, June 1994, pp.18-19.
- (Ko97) Koenig, A. and B.Moo, *Ruminations on C++*, Reading, MA: Addison-Wesley, 1997.
- (Kr91) Kruse, R.L.; B.P.Leung; and C.L. Tondo, *Data Structures and Program Design in C*, Englewood Cliffs, NJ: Prentice Hall, 1991.
- (Le92) Lejter, M.; S. Meyers; and S.P. Reiss, "Support for Maintaining Object-Oriented Programs," *IEEE Transactions on Software Engineering*, Vol.18, No.12, December 1992, pp.1045-1052.
- ((Li91) Lippman, S.B., *C++ Primer* (Second Edition), Reading, MA: Addison-Wesley Publishing Company, 1991.
- (Lo93) Lorenz, M., *Object-Oriented Software Development: A Practical Guide*, Englewood Cliffs, NJ: Prentice Hall, 1993.
- (Lo94) Lorenz, M., "A Brief Look at Inheritance Metrics," *The Smalltalk Report*, Vol. 3, No.8 June 1994, pp. 1, 4-5.
- (Ma93) Martin, J., *Principles of Object-Oriented Analysis and Design*, Englewood Cliffs, NJ: Prentice Hall, 1993.
- (Ma95) Martin, R. C., *Designing Object-Oriented C++ Applications Using the Booch Method*, Englewood Cliffs, NJ: Prentice Hall, 1995.
- (Ma93a) Matsche, J.J., "Object-Oriented Programming in Standard C," *Object Magazine*, Vol.2, No. 5, January/February 1993, pp. 71-74.
- (Mc94) McCabe, T.J., and A.H.Watson, "Combining Comprehension and Testing in Object-Oriented Development," *Object Magazine*, Vol.4, No. 1, March/April 1994, pp. 63-66.
- (Me88) Meyer, B., *Object-Oriented Software Construction*, C.A.R.Hoare Series Editor, Englewood Cliffs, NJ: Prentice Hall, 1988.

- ( Me92 ) Meyer, B., *Advances in Object-Oriented Software Engineering*, Edited by D. Mandrioli and B. Meyer, Englewood Cliffs, NJ: Prentice Hall, 1992.
- ( Me92a ) Meyer, B., *Eiffel: The Language*, Englewood Cliffs, NJ: Prentice Hall, 1992.
- ( Me92b ) Meyers, S., *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Reading, MA: Addison-Wesley Publishing Company, 1992.
- ( Me95 ) Meyers, S., *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Reading, MA: Addison-Wesley Publishing Company, 1995.
- ( Me95a ) Meyers, S., "Mastering User-Defined Conversion Functions," *C/C++ Users Journal*, Vol. 13, No. 8, August 1995, pp. 57-63.
- ( Mu93 ) Murray, R., *C++ Strategies and Tactics*, Reading, MA: Addison-Wesley Publishing Company, 1993.
- ( Mu94 ) Musser, D. R., and A. A. Stepanov, "Algorithm-Oriented Generic Libraries," *Software Practice and Experience*, Vol. 24, No. 7, July 1994.
- ( Mu96 ) Musser, D. R., and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Reading, MA: Addison-Wesley Publishing Company, 1996.
- ( Ne95 ) Nelson, M., *C++ Programmer's Guide to the Standard Template Library*, Foster City, CA: Programmers Press, 1995.
- ( Ne920 ) Nerson, J. M., "Applying Object-Oriented Analysis and Design," *Communications of the ACM*, Vol. 35, No. 9, September 1992, pp. 63-74.
- ( Ni92 ) Nierstrasz, O.; S. Gibbs; and D. Tsichritzis, "Component-Oriented Software Development," *Communications of the ACM*, Vol. 35, No. 9, September 1992, pp. 160-165.
- ( Pi90 ) Pinson, L. J., and R. S. Wiener, *Applications of Object-Oriented Programming*, Reading, MA: Addison-Wesley, 1990.
- ( Pi93 ) Pittman, M., "Lessons Learned in Managing Object-Oriented Development," *IEEE Software Magazine*, Vol. 10, No. 1, January 1993, pp. 43-53.
- ( Pl92 ) Plauger, P. J., *The Standard C Library*, Englewood Cliffs, NJ: Prentice Hall, 1992.
- ( Pl93 ) Plauger, D., "Making C++ Safe for Threads," *The C Users Journal*, Vol. 11, No. 2, February 1993, pp. 58-62.
- ( Po97 ) Pohl, I., *C++ Distilled: A Concise ANSI/ISO Reference and Style Guide*, Reading, MA: Addison-Wesley, 1997.
- ( Po97a ) Pohl, I., *Object-Oriented Programming Using C++*, Second Edition, Reading, MA: Addison-Wesley Publishing Company, 1997.
- ( Pr92 ) Press, W. H., et al, *Numerical Recipes in C*, Second Edition, Cambridge, MA: Cambridge University Press, 1992.
- ( Pr93 ) Prieto-Diaz, R., "Status Report: Software Reusability," *IEEE Software*, Vol. 10, No. 3, May 1993, pp. 61-66.
- ( Pr92 ) Prince, T., "Tuning Up Math Functions," *The C Users Journal*, Vol. 10, No. 12, December 1992.
- ( Pr95 ) Prose, J., "Wake Up and Smell the MFC: Using the Visual C++ Classes and Applications Framework," *Microsoft Systems Journal*, Vol. 10, No. 6, June 1995, pp. 17-34.

- ( Ra90 ) Rabinowitz, H., and C. Schaap, *Portable C*, Englewood Cliffs, NJ: Prentice Hall, 1990.
- ( Re91 ) Reed, D.R., "Moving from C to C++," *Object Magazine*, Vol.1, No.3, September/October 1991, pp.46-60.
- ( Ri78 ) Ritchie, D.M.; S. C. Johnson; M.E. Lesk; and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," *The Bell System Technical Journal*, Vol. 57, No.6, Part 2, July-August 1978, pp.1991-2019.
- ( Ri84 ) Ritchie, D.M., "The UNIX System: The Evolution of the UNIX Time-Sharing System," *AT&T Bell Laboratories Technical Journal*, Vol.63, No.8, Part 2, October 1984, pp.1577-1593.
- ( Ro84 ) Rosler, L., "The UNIX System: The Evolution of C-Past and Future," *AT&T Laboratories Technical Journal*, Vol.63, No. 8, Part 2, October 1984, pp. 1685-1699.
- ( Ru92 ) Rubin, K.S. and A. Goldberg, "Object Behavior Analysis," *Communications of the ACM*, Vol. 35, No. 9, September 1992, pp. 48-62.
- ( Ru91 ) Rumbaugh, J.; M. Blaha; W. Premerlani; F.Eddy; and W. Lorensen, *Object-Oriented Modeling and Design*, Englewood Cliffs, NJ: Prentice Hall, 1991.
- ( Sa93 ) Saks, D., "Inheritance," *The C Users Journal*, May 1993, pp.81-89.
- ( Se92 ) Sedgwick, R., *Algorithms in C++*, Reading, MA: Addison-Wesley, 1992.
- ( Se92a ) Sessions, R., *Class Construction in C and C++*, Englewood Cliffs, NJ: Prentice Hall, 1992.
- ( Sk93 ) Skelly, C., "Pointer Power in C and C++," *The C Users Journal*, Vol. 11, No. 2, February 1993, pp. 93-98.
- ( Sm92 ) Smaer, S. and S. J. Mellor, *Object Lifecycles: Modeling the World in States*, Englewood Cliffs, NJ: Yourdon Press, 1992.
- ( Sm90 ) Smith, J. D., *Reusability & Software Construction in C & C++*, New York, NY: John Wiley & Sons, 1990.
- ( Sn93 ) Snyder, A., "The Essence of Objects: Concepts and Terms," *IEEE Software Magazine*, Vol.10, No.1, January 1993, pp. 31-42.
- ( St95 ) Stepanov, A. and M. Lee, "The Standard Template Library," Internet Distribution, Published at [ftp:// butler.hpl.hp.com/stl](ftp://butler.hpl.hp.com/stl), July 7, 1995.
- ( St84 ) Stroustrup, B., "The UNIX System: Data Abstraction in C," *AT&T Bell Laboratories Technical Journal*, Vol.63, No.8, Part 2, October 1984, pp.1701-1732.
- ( St88 ) Stroustrup, B., "What is Object-Oriented Programming?" *IEEE Software*, Vol. 5, No. 3, May 1988, pp.10-20.
- ( St88a ) Stroustrup, B., "Parameterized Types for C++," *Proceedings of the USENIX C++ Conference*, Denver, CO, October 1988.
- ( St91 ) Stroustrup, B., *The C++ Programming Language* ( Second Edition ), Reading, MA: Addison-Wesley Series in Computer Science, 1991.
- ( St93 ) Stroustrup, B., "Why Consider Language Extensions?: Maintaining a Delicate Balance," *C++ Report*, September 1993, pp.44-51.
- ( St94 ) Stroustrup, B., "Making a vector Fit for a Standard," *The C++ Report*, October 1994.
- ( St94a ) Stroustrup, B., *The Design Evolution of C++*, Reading, MA: Addison-Wesley Publishing Company, 1994.

- ( St97 ) Stroustrup, B., *The C++ Programming Language*, Third Edition, Reading, MA: Addison Wesley Publishing Company, 1997.
- ( Ta94 ) Taligent Inc., *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*, Reading, MA: Addison-Wesley Publishing Company, 1994.
- ( Ta92 ) Taylor, D., *Object-Oriented Information Systems*, New York, NY: John Wiley & Sons, 1992.
- ( To89 ) Tondo, C.L. and S.E.Gimpel, *The C Answer Book*, Englewood Cliffs, NJ: Prentice Hall, 1989.
- ( Ur92 ) Urlocker, Z., "Polymorphism Unbounded," *Windows Tech Journal*, Vol. 1, No. 1, January 1992, pp.11-16.
- ( Va95 ) Van Camp, K.E., "Dynamic Inheritance Using Filter Classes," *C/C++ Users Journal*, Vol. 13, No.6, June 1995, pp.69-78.
- ( Vi94 ) Vilot, M.J., "An Introduction to the Standard Template Library," *The C++ Report*, Vol.6, No.8, October 1994.
- ( Vo91 ) Voss, G., *Object-Oriented Programming: An Introduction*, Berkeley, CA: Osbourne McGraw-Hill, 1991.
- ( Vo93 ) Voss, G., "Objects and Messages," *Windows Tech Journal*, February 1993, pp. 15-16.
- ( Wa94 ) Wang, B.L. and J.Wang, "Is a Deep Class Hierarchy Considered Harmful?" *Object Magazine*, Vol.4, No.7, November-December 1994, pp.35-36.
- ( We94 ) Weisfeld, M., "An Alternative to Large Switch Statements," *The C Users Journal*, Vol. 12, No.4, April 1994, pp.67-76.
- ( We92 ) Weiskamp, K. and B. Flamig, *The Complete C++ Primer*, Second Edition, Orlando, FL: Academic Press, 1992.
- ( Wi93 ) Wiebel, M. and S.Halladay, "Using OOP Techniques Instead of *switch* in C++," *The C Users Journal*, Vol.10, No.10, October 1993, pp.105-112.
- ( Wi88 ) Wiener, R.S. and L.J.Pinson, *An Introduction to Object-Oriented Programming and C++*, Reading, MA: Addison - Wesley, 1988.
- ( Wi92 ) Wilde, N., and R. Huitt, "Maintenance Support for Object-Oriented Programs," *IEEE Transactions on Software Engineering*, Vol.18, No.12, December 1992, pp. 1038-1044.
- ( Wi93 ) Wilde, N.; P. Matthews; and R. Huitt, , "Maintaining Object-Oriented Software," *IEEE Software Magazine*, Vol.10, No.1, January 1993, pp. 75-80.
- ( Wi96 ) Wilson, G.V. and P.Lu, *Parallel Programming Using C++*, Cambridge, MA: MIT Press, 1996.
- ( Wi93 ) Wilt, N., "Templates in C++," *The C Users Journal*, May 1993, pp. 33-51.
- ( Wi90 ) Wirfs-Brock, R.;B. Wilkerson; and L. Wiener, *Designing Object-Oriented Software*, Englewood Cliffs, NJ: Prentice Hall, 1990.
- ( Wy92 ) Wyatt, B.B.; K. Kavi; and S. Hufnagel, "Parallelism in Object-Oriented Languages: A Survey," *IEEE Software*, Vol.9, No.7, November 1992, pp.56-66.
- ( Ya93 ) Yamazaki, S.; K. Kajihara; M. Ito; and R. Yasuhara, "Object-Oriented Design of Telecommunication Software," *IEEE Software Magazine*, Vol.10, No.1, January 1993, pp. 81-87.